Sri Lankan Journal of Technology
Research

SLJoT
2026 Issue 01 13-26
ISSN-2773-6970

Hybrid Test Automation Framework for Graphical User Interface

M.S.Fathima Fayaza

Department of Information Technology

Corresponding Author: fayaza@seu.ac.lk || ORCID: 0000-0002-3397-7999

Received: 30-06-2026. * Accepted: 23-04-2026 * Published Online: 30-06-2026

Abstract – Modern software development is greatly aided by automated software testing that provides continuous feedback on product quality. Software testing is crucial since the system's complexity increases daily with requirement changes and technological advancements. The Graphical User Interface (GUI) is the most typical interface to interact with the software. Around 60% of the overall code is related to GUI in software. Many solutions have been developed in response to the increased demand for User Interface (UI) test automation. Researchers and developers have been working tirelessly to improve upon existing methods. The GUI testing process has been automated in various ways, such as recording and playback tools, keyword-driven methodology, model-based approaches and hybrid models. A small change in the locations of the GUI elements needs many updates in the testing scripts. This paper introduced a Hybrid Test Automation Framework using selenium to overcome challenges in the GUI functions and property changes. The framework was tested in two versions of the e-commerce web application: E-commerce web application with a unique UI object ID, and E-commerce web application with traditional implementation. An e-commerce web application with a unique UI object ID performs better than a traditional implementation.

Keywords- Automation Framework, Selenium, Software Testing, Test GUI Automation

Recommended APA Citation

Fayaza, M. S. F. (2026). Hybrid test automation framework for graphical user interface. *Sri Lankan Journal of Technology, 7*(1), 13–26.

Introduction

Software testing is one of the key steps in the software development process. Testing is used to validate and verify software product quality (Wiklund et al., 2017). Around 30% - 80% of the cost, 40%-50% of resources and 30% of total efforts are related to testing (Kumar & Mishra, 2016; Wiklund et al., 2017). Testing is a major challenge in software development and optimizing the testing process can reduce the cost and time to market (Pelivani & Cico, 2021). Software testing can be done in two ways: manual testing and automated testing (Pelivani & Cico, 2021). Manual testing uses human resources to test the system. Automated testing uses standard software solutions to control the execution of test cases of software/system under test (SUT). In manual testing, the number of executions defines the testing cost, but in automation, the number of test cases that need to be automated defines the testing cost (Hanna, Elsayed, et al., 2018). In test automation, the main cost occurs for implementing and maintaining the automation system (Gojare et al., 2015). However, in the real world, manual test execution is carried out with variations of the same test case.

Industries use test automation for smoke, regression, performance, load, and application programming interfaces (APIs) testing (Guo et al., 2010). Regression risks and issues can be mitigated by automating test cases. Between 60% and 90% of the bugs are caused by 20% of the modules, with a median of around 80% (Ramler & Wolfmaier, 2006). Test automation helps bring down the cost, time and effort of software testing (Kuusinen et al., 2017). Automation helps increase test coverage efficiency, speed up execution and reduce human errors (Pelivani & Cico, 2021). Further, in agile practice, automation plays a vital role in developing and testing synchronization (Kuusinen et al., 2017). Automation enables test teams to quickly and accurately prepare test data, run complicated test cases across numerous platforms, and detect flaws early in the development cycle (Pelivani & Cico, 2021). Modern test automation systems assist enterprises in various ways; when implemented effectively, they can increase test team productivity, improve software quality, and reduce time to market (Hanna, Elsayed, et al., 2018). The most common way to interact with the software is through a graphical user interface (GUI). The GUI responds to user actions, such as mouse clicks and keystrokes. The user can communicate with the underlying application using GUI through method calls or a message-passing mechanism. A significant percentage of the code is dedicated to implementing and maintaining the GUI. According to research, GUI implementation accounts for up to 60% of overall software code (Nass et al., 2021; Pradhan, 2011). The proper testing of the GUI can make a big difference in the application's overall safety, robustness, and usefulness (Nass et al., 2021).

Most of the studies in the literature are connected to test automation tool selection (Gamido & Gamido, 2019; Guo et al., 2010; Pelivani & Cico, 2021), analysis of the test automation benefits and outcomes (Grater, 2005; E. H. Kim et al., 2009; Rafi et al., 2012; Ramler & Wolfmaier, 2006). However, there are many technical challenges in implementing the automation for GUI (Nass et al., 2021). Nass et al. (Nass et al., 2021) categorize these challenges as essential to accidental difficulties. If the challenge is innate to specific technology or approach, it is essential and if the challenge can eliminate it is accidental. “Application changes break test execution”, “tool challenges”, “test automation of dynamic applications” and “state space explosion during model-based testing (MBT)” are essential challenges (Nass et al., 2021). “Synchronization/ timing between test and SUT”, “robust identification of GUI widgets”, “Requires automation or programming skills”, and “Creating /maintaining model-based test” are accidental (Nass et al., 2021). Eliminating essential challenges is difficult, however accidental challenges can be minimized (Nass et al., 2021).

Lack of vigorous detection of GUI widgets is a base for synchronization challenge (Nass et al., 2021; Pradhan, 2011). Further, frequent software updates demand frequent modification and update in the automation script. This is very time-consuming and cost-consuming work.

Researchers and industry have put in several efforts to overcome these challenges, but until today there is no full-fledged mechanism to address this issue (Nass et al., 2021). Further, 45% of automation engineers are unsatisfied with existing automation frameworks since they lack the features they expect for their projects (Nass et al., 2021).

Even though there are many frameworks introduced, high maintenance costs of test scripts, poor reusability of test scripts, and difficulty in handling dynamic GUI remain challenges (Abdalftah Fadul Mohammed & Ahmed Ibrahim, 2025). The core of all these challenges is the GUI locator changes and functional changes in SUT. To overcome these challenges this study introduces the hybrid test automation framework using selenium to overcome the obstacles in GUI automation. The researcher presents constant unique IDs to identify the objects on the web page and uses the page object model to save the object on a web page. Further, the framework uses cache mechanisms to avoid duplication of work in test execution. Moreover, this framework takes care of automatic integration, version control, test execution and test report generation. The framework was tested on e-commerce applications developed by researchers.

Literature Review

In literature researchers analyze the impact of test automation (Kumar & Mishra, 2016), benefits and challenges (Nass et al., 2021; Wiklund et al., 2017), and technical perspectives and the outcomes of different technical approaches (Vila et al., 2017). The finding shows that even though there are many advantages of test automation, there are still some challenges in GUI test automation. For example, “changed screen resolution”, “fails for unknown reason”, “non-determinism”, “sensitivity to business logic”, “cascading error”, “environment/hardware configuration”, “high maintenance cost of test cases”, “long setup time”, “HMI must be available”, “hard to reproduce the error”, “less effective in detecting faults”, “requires workflow changes”, “low test coverage during MBT”, “identify less faults when using an oracle”, “limited applicability of models” and “slow test execution during MBT” are some academically reported as GUI test automation challenges in the literature (Nass et al., 2021). The key reasons for these challenges are the continuing evolution in the SUT, GUI updates in the system, challenges in uniquely identifying the GUI elements, synchronization issues, and environmental complexity. By analyzing literature through a survey study, Wiklund et al. (2017) concluded that a successful test automation demands obstacles connected to test automation that are prevented and removed quickly (Wiklund et al., 2017).

The researchers analyze the impact of test automation in the cost (Hanna, Elsayed, et al., 2018; Kumar & Mishra, 2016; Ramler & Wolfmaier, 2006) time to market (Kumar & Mishra, 2016) and quality perspectives (Kumar & Mishra, 2016), where all the factors are positive impacts on the outcome. Kumar and Mishra (Kumar & Mishra, 2016) analyze the cost impact using three different products and concluded that applying test automation positively impacts the cost. Further, they stated that creating and maintaining a test case is expensive but more profitable in the long run (Kumar & Mishra, 2016). Ramler and Wolfmier (Ramler & Wolfmaier, 2006) discussed the economic viewpoint in test automation using the opportunity cost by answering the question, "When should a test be automated?". The opportunity cost of automating one test case equals four manual test executions (Ramler & Wolfmaier, 2006). The cost of automating is affected by the number of test cases, while the cost of manual testing is affected by the number of test executions. As a result, the number of test executions significantly influences cost and time in manual testing, whether testing the unique test cases or same test case twice or multiple times. Industrial experts form their experience and knowledge gain, point out: I. each test case should be unique and independent because that would prevent failure due to the dependencies, II. knowing the development techniques helps speed up the automation

process and III. all test and framework codes need to preserve equal importance (Debroy et al., 2018).

Vila et al. in (2017) discussed the opportunities and challenges in web application testing with selenium. Authors found building a test automation framework improves testing quality, flexibility, and reusability (Vila et al., 2017). Also, there are opportunities to expand the framework for handling standard functionalities (Vila et al., 2017). Hanna et al. (Hanna, Aboutabl, et al., 2018) proposed a test automation framework for web applications that was capable of saving 75% of time and effort involved in the automation. The framework performs well, 21% more efficient than using selenium IDE (Hanna, Aboutabl, et al., 2018).

Gojare et al. (2015) introduced an automation framework using selenium web driver (*SeleniumHQ Browser Automation*, n.d.). Since the selenium web driver did not support the test report generation feature, researchers integrated TestNG (*TestNG - Welcome*, n.d.) to generate the test result report. Another framework support customised test report emails and allowed screenshots for failed test cases. Wang and De (Wang & Du, 2012) proposed a framework using JMeter and Selenium. JMeter is used to create different load types on the SUT. Further, this framework supports backend testing and multiple browsers supporting. Here researchers separated the test data from the flow. Therefore, the same data can be used for different types of testing, such as smoke, regression, etc.

Kim et al. (2007) introduced a test automation framework called NTAF for distributed environments, where they integrated continuous integration with FitNesse (*FrontPage*, n.d.). Also, the NTAF supports stress and performance tests, which are challenging to perform manually. Further, NTAF supports automated builds, bug tracking, testing, metrics measurement, and concurrent detection (D. E. Kim & Park, 2007).

Pajunen et al. (2011) introduced the keyword-driven test automation framework. The keywords-driven approach helps improve and maintain related test cases and is easy to build. Keyword-driven test cases are shorter than other test cases. MBT creates a model for the system under test (*Robot Framework*, n.d.). In this approach, the Robot Framework is integrated with the TEMA toolset to create MBT. Kim et al. (2009) proposed an automation framework for agile development. The framework is based on fitNesse and supports continuous integration. Agile support helps with fast delivery.

Manual debugging is time-consuming and error-prone and to overcome this challenge an automated debugging HTML tool was introduced (Mahajan & Halfond, 2015). This tool uses a computer vision-based algorithm and compares the browser-rendered image with its oracle image (Mahajan & Halfond, 2015). Test automation demands human intervention to create test scripts. To overcome this challenge researchers introduced automation script generation from manual English-written test cases (Thummalapenta et al., 2012). For this, the authors used a backtracking-based search to identify the ambiguities (Thummalapenta et al., 2012). This approach reduced the cost of automation very much and the proposed framework achieved 82% accuracy (Thummalapenta et al., 2012).

Test Automation Frameworks

This section discusses the test automation frameworks available in the literature.

Record/playback automation

This approach does not need programming skills and is a fundamental approach. User actions are recorded using the record button, and auto-generated scripts are played using the play button (Hanna, Elsayed, et al., 2018). This is easy to use and no coding is required. The major drawback

of this approach is more sensitive to small changes (Hanna, Elsayed, et al., 2018) and poor maintainability.

Data-driven framework

Data-driven frameworks separate the test data from the test script. Therefore, it is supported to test the same functionality with different datasets (Pelivani & Cico, 2021), (Mishra & Atesogullari, 2020). These approaches need programming skills and it is hard to handle large external datasets.

Keyword-driven framework

These frameworks save keywords in a separate file from the data and test script. Most of the time, test data and keywords are kept like a table structure (Pelivani & Cico, 2021). Maintenance becomes a challenge with the keyword library expanding.

Modular framework

This is based on the object-oriented programming concept. These frameworks divide the SUT into unique modules. By combining modules, scripts are generated for testing (Pelivani & Cico, 2021). Scalability is the key challenge that remains in this approach.

Hybrid framework

By combing multiple approaches, these frameworks take advantage of all the approaches (Pelivani & Cico, 2021). However, the complexity and integration overhead increases.

Test Automation Framework Design and Implementation

The first step in the automation framework building process is automation tool selection based on the SUT requirements. Automation tools can be open-source or commercial. Open-source tools are free, while commercial tools provide more advanced features, and the license cost is high. The researcher selected the free, open-source Selenium (*SeleniumHQ Browser Automation*, n.d.) as the base tool for this study. Selenium needs more programming ground and demands minimum maintenance (Dobslaw et al., 2019). Further, Selenium provides extensive cross browser support, language flexibility, and facilitates integration with external tools and extensions which makes it best for hybrid framework design. However, the hybrid nature of this framework introduces some additional implementation complexity compared to standalone solutions such as selenium-based scripting.

There are numerous approaches to automate the testing of the source code, but GUI testing is still challenging with continuously updating GUI. However, with the growing use of GUI to interact with software systems, it has become clear that GUI testing is an important stage in software verification and validation. The main challenge in GUI automation is that every small change in the application needs greater effort to update the automation script. For example,

I. the previous web application version had the 'Register' button, which was changed to the 'New User' button in the new version,
II. GUI element locator change.

Fig. 1 shows the example of GUI placeholder Attribute change. The most common GUI changes are:

1. GUI locator changes

2. GUI function and execution flow changes.

To overcome these challenges, this study introduced a hybrid test automation framework. The framework aims to gain advantages from multiple approaches such as data driven and modular. Fig. 2 illustrates the high-level architecture of the system. This framework consists of two layers:

- Presentation layer
- Automation Testing Framework

Presentation layer:

The Graphical User Interface is the core of the presentation layer. The Quality Assurance Engineers use GUI to communicate with the system. GUI allows the user to create, edit, execute test scripts, and check the execution result report.

Automation Testing Framework

The framework is responsible for 1. Continuous Integration, 2. Version control, 3. Test automation, 4. Test execution and 5. Test result report generation.

The framework consists of following modules.

I. Version control
II. CI /CD server
III. Automation Framework Engine
IV. Automation Framework Utils
V. Automation Framework API
VI. Test Report

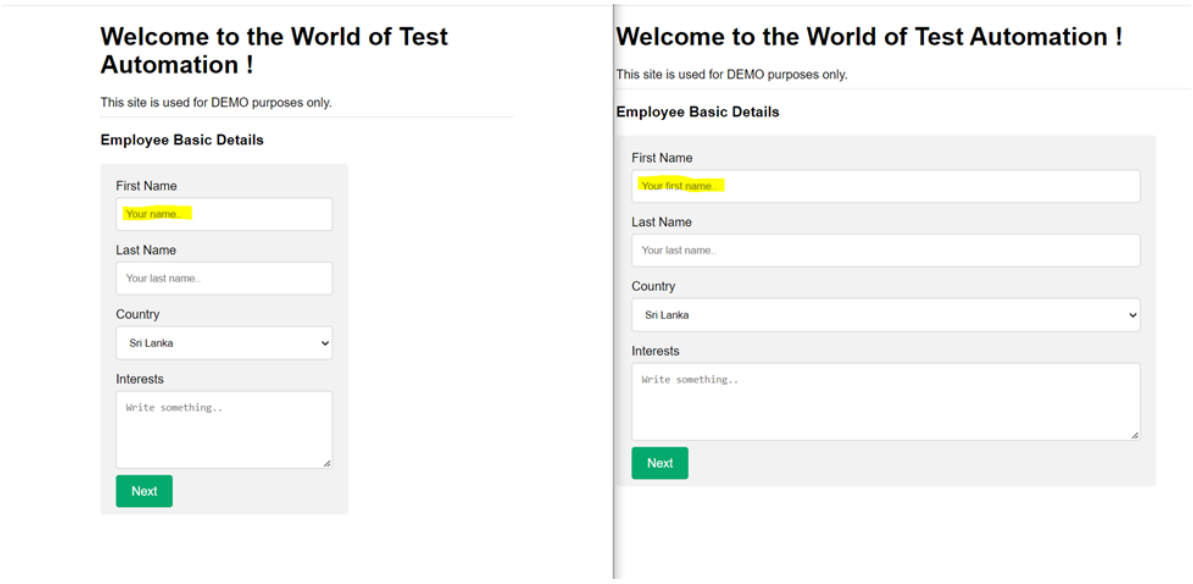

Figure 1. *Sample GUI change in the SUT*

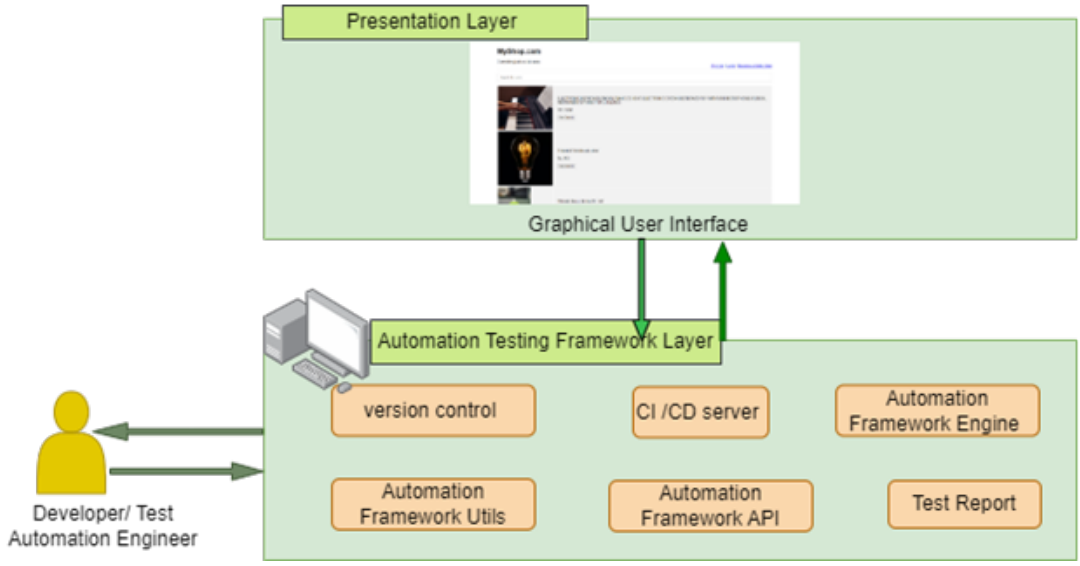

Figure 2. *High-level architecture of the Test Automation Framework*

I. **Version control:** This framework supports version control using GitHub (*GitHub: Let's Build from Here · GitHub*, n.d.). This helps test multiple versions of SUT and keep various versions of the automation script. This helps track the changes in the SUT and automation scripts.

II. **CI/CD server:** Multiple automation engineers work simultaneously in the test automation framework. Before the execution, the framework must integrate the changes. For the integration, this framework uses Jenkins(Jenkins, n.d.).

III. **Automation Framework Engine:** This is the core module of the framework. This uses Selenium to perform testing. Selenium allows mimicking the user actions such as mouse clicks and keyboard inputs. The framework has the following structure to increase usability of engine:

Configuration: Framework saves the configuration details under the config folder. All the configurations are saved inside the configuration.properties file. This contains the SUT context, user context (username, password), environment context (browser, browser version), platform context and extension details. This helps run the execution in multiple browsers with different versions of applications with different configurations.

Functions: The most significant challenge in GUI testing is GUI flow changes. Requirements change or technological advancements can drive these changes. For example, in a supply chain application, the old version of the application is not allowed to amend the purchase order, but in the new version, the purchase order amendment is permitted. This functional change is the result of the requirements change in the SUT. GUI locator changes, system logic changes, requirement changes, and adding or updating the features are some causes for GUI flow changes that lead to updating the script. The best way to overcome this challenge is to introduce the functions for unique flows and maintain related functions in separate classes and create the script

with function calls. If the flow needs to be updated, it's enough to update the function in a single place rather than updating it in multiple files.

Test Data: For testing, testers need to input some values. For example, the system login is required to provide the username and password for the testing. This framework is used to separate the test data from the script. This helps to test the same feature with different datasets. Further, it increases the test coverage and helps to cover the boundary values.

Preconditions: In real world, to run some test cases you need to execute some other test cases or functions as preconditions. For example, in some systems, creating an invoice requires a purchase order number. In this example, the purchase order number is a precondition for invoice creation. Sometimes, one test case can be a precondition for another test case. In such situations, some selected test cases execute multiple times. To overcome this challenge this study introduces a cache mechanism with test plan execution to avoid duplicate execution. The Cache can be used to save the object needs for future execution.

IV. **Automation Framework Utils:** This contains the core functions and libraries of the framework.

V. **Automation Framework API:** This contains the page object classes needed for automation.

Object repository: The positioning of elements is the most critical aspect of User Interface (UI) automation. Depending on the automation tool used, it's known as find strategy, lookup, control identification, XPath, etc. Simply, the locator helps identify objects on the page. Some of the UI locator issues leading to automation failure are:

- Target element moves on the UI (Location change)
- UI changes (new elements, elements deleted, elements moved, etc.)
- The data changes in UI (new rows appear on a table, sorting orders change, etc.)
- Locator dynamically generates

UI locators are affected by various factors such as system design, the technology used for the system, UI design, development, etc. Page object identification mechanisms play a critical role in the success of an automation project. Locators can save in scripts, dictionaries, static classes, or page object models. UI locators can save as absolute locators or relative locators. Commonly used locator types are 1. ID, 2. Cascading Style Sheet (CSS), 3. XPath and 4. Text content. But all these approaches have their challenges, and when an UI locator change happens, most of the test cases fail. To overcome these, this study introduces unique constant IDs or objects in UI from the system design level. Unique constant ID is a simple and effective way to identify the unique elements or objects in UI. Commonly, developers are used to having the same ID for multiple objects in the UI or dynamic ID for UI elements. Also, this paper suggests having a Page object model as an object repository. This helps update the changes in a single place rather than updating every script.

VI. **Test Report:** The framework supports parallel and sequential execution using a selenium grid and at the end of the execution framework generates the test report. The framework supports single test case execution to test plan execution.

Test plan: When the number of test cases increases, the execution time also increases. For example, regression test cases. To overcome this, the researcher introduced separate test plans for each and every test. For that, first test engineers must analyze the STU and need to group the related test cases. For example, Smoke test plan, regression test plan.

Empirical Evaluation: Framework Testing

The framework was empirically evaluated using web-based e-commerce applications created by researchers to ensure reproducibility. This application contains product management, customer management, and order tracking features. The researcher created two versions of web-based e-commerce applications for testing.

I. E-commerce web application without unique UI object id (base system)
II. E-commerce web application with unique UI object id

Framework was used to test user interface changes and functional changes of SUT with support of professional test automation engineer. Researchers created forty-eight test cases covering all three functional areas of system. This covers the 95 percentage of GUI elements and functional paths. Fifteen (15) test cases based on Customer Management module, twenty (20) test cases based on Product Management module and thirteen (13) test cases based on Order tracking module. All the test cases were evaluated using professional software quality assurance engineers.

User Interface modification

Some modifications were introduced to test the framework in both versions of SUT. For example, I. the “Register Now” button changes into the “SignUp” button, II. Xpath of some page object is changed.

Table I shows the execution result of both versions of SUT for page layout changes Where 92 percent of test cases got pass application with the unique UI object ID and only 33 percent of testcases got pass with base application. GUI page layout changes heavily affect the base application more than the application with the unique UI object ID. It is clearly shown that the Unique Id is constant on a web page and even though the layout changed the Id remained the same, and therefore failures were very low when the GUI got changed. Fig. 3. shows the execution output for the layout changes. Table II shows the impact of GUI navigation path changes. A total of 62.5 percent of test cases got passed with the application with unique UI object id while only 16 percent of test cases got passed with baseline application. Further, the number of failed and skipped test cases are higher in the base system than System with unique Id. This is because Id is unique and even though the path changed, Id remained unchanged.

Modify the function of a system under test

Adding new steps or removing new steps modifies the existing function of STU. For example, the old version of SUT allows to create a shipping order without adding an invoice number and the new version of SUT requesting an invoice number to create a shipping order. As per this example, existing functions need to update with the new step. Table III shows the execution

result of the functional modification of a system. In the framework, the SUT unique functions are saved separately rather than added as steps in the scripts. When a function gets changed, modifying a single place can fix the failures. This helps to reduce the cost of time for maintenance. Figure 4 shows the automation result after the SUT function modification. The result shows that the framework is more reliable for after release work.

Table 1. *The execution output result of page layout changes*

Functional Area	**Application Without Unique Id (Base System)**			**Application With Unique Id**		
	Pass	***Skipped***	***Failed***	***Pass***	***Skipped***	***Failed***
Customer management test cases (15)	5	4	6	14	0	1
Product management test cases (20)	8	4	8	18	2	0
Order tracking test cases (13)	2	1	10	12	0	1

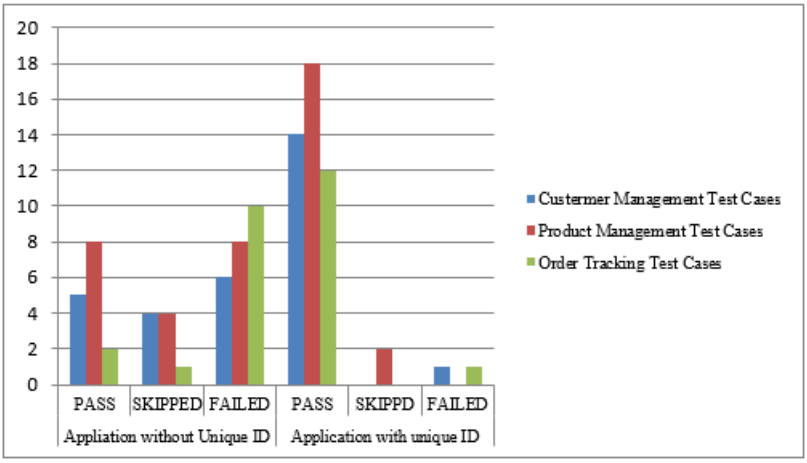

Figure 3. *Page layout changes impact on the test result*

Table 2. *The execution output result of navigation path changes*

Functional Area	**Application Without Unique Id (Base System)**			**Application With Unique Id**		
	Pass	**Skipped**	**Failed**	**Pass**	**Skipped**	**Failed**

Customer Management Test Cases (15)	3	5	7	10	3	2
Product Management Test Cases (20)	2	8	10	12	2	6
Order Tracking Test Cases (13)	3	6	4	8	2	3

Table 3. *The execution output result of modify the function of a system under test*

FUNTIONAL AREA	**Application Without Unique ID**			**Application With Unique ID**		
	Pass	**Skipped**	**Failed**	**Pass**	**Skipped**	**Failed**
Customer Management Test Cases (15)	3	2	10	14	0	1
Product Management Test Cases (20)	3	1	15	18	2	
Order Tracking Test Cases (13)	3	0	10	12	0	1

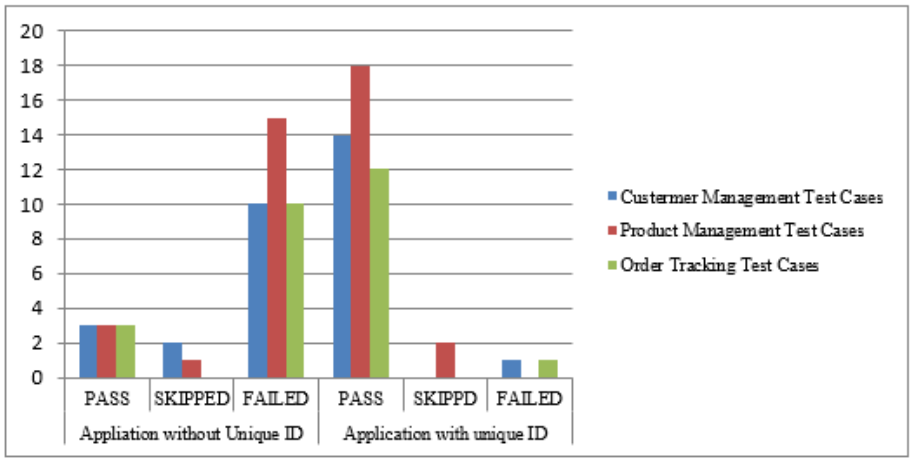

Figure 4. *Functional changes impact the test result*

Discussion

The existing GUI test automation frameworks provide significant improvement in the cost, time to market and quality of the SUT. However, with continued evolution of SUT and GUI changes create persistent challenges.Introducing a unique Id for GUI elements reduces the failures by the GUI elements' position or location changes. Also, spreading test data from test script and creating unique functions for separate related tasks and the page object model increases the usability and maintenance of the system.

When the number of test cases increases, the time taken for execution also increases. These become challenges when the number of test cases is high. For example, running the regression testing. Also, sometimes some test cases have dependencies and prerequisites so that running the same test case multiple times is a waste of time and money. Therefore, when designing the test plan, test engineers need to consider the system level and analyze to create the test plan.

In the real world, practitioners start testing after the implementation of the system. However, the findings of this study emphasizes that test automation needs to be started from the requirements specification level to get the maximum output. Giving the correct idea about the automation to different stakeholders such as the system analysis, architectural design, and development team, will increase the outcomes of the test automation.

Conclusion

In software development, software testing has become a very critical and time-consuming step. Test automation has been introduced to overcome manual testing difficulties. However, Graphical User Interface (GUI) test automation still faces many challenges with the continuous evolution of System Under Test (STU) and GUI changes. To mitigate these challenges, this paper discusses the hybrid test automation framework with Version control, CI /CD server, Automation Framework Engine, Automation Framework Utils, Automation Framework API and Test Report modules. In this study, the author introduces a unique constant ID for each and every element in the GUI, page object model, and separated data and script increase the useability, maintainability performance. The study tested the framework using the e-commerce web application with and without unique UI object id. The output shows that framework outperforms with e-commerce web application with unique UI object id. This study suggests that automation processes need to start with the Software Requirements Specification (SRS) to get the maximum benefit in the software development life cycle.

Acknowledgment

The author thanks Mr. Sajitha Pathirana for his valuable insight and support in the testing process.

References

Abdalftah Fadul Mohammed, A., & Ahmed Ibrahim, K. (2025). Challenges in GUI test automation for dynamic web applications: A systematic review. *Excellence Journal for Engineering Sciences, 2*(5).

Debroy, V., Brimble, L., Yost, M., & Erry, A. (2018). Automating web application testing from the ground up: Experiences and lessons learned in an industrial setting. Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 354–362. https://doi.org/10.1109/ICST.2018.00042

Dobslaw, F., Feldt, R., Michaelsson, D., Haar, P., Neto, F. G. de O., & Torkar, R. (2019). Estimating return on investment for GUI test automation tools. arXiv. http://arxiv.org/abs/1907.03475

FrontPage. (n.d.). Retrieved June 21, 2021, from http://fitnesse.org/FrontPage

Gamido, H. V., & Gamido, M. V. (2019). Comparative review of the features of automated software testing tools. International Journal of Electrical and Computer Engineering, 9(5), 4473–4478. https://doi.org/10.11591/ijece.v9i5.pp4473-4478

GitHub. (n.d.). Retrieved December 29, 2022, from https://github.com/

Gojare, S., Joshi, R., & Gaigaware, D. (2015). Analysis and design of Selenium WebDriver automation testing framework. Procedia Computer Science, 50, 341–346. https://doi.org/10.1016/j.procs.2015.04.038

Grater, M. T. (2005). Benefits of using automated software testing tools to achieve software quality assurance. https://scholarsbank.uoregon.edu/xmlui/handle/1794/7817

Guo, W., Fu, X., & Feng, J. (2010). Integration system.

Hanna, M., Aboutabl, A. E., & Mostafa, M.-S. M. (2018). Automated software testing framework for web applications. International Journal of Applied Engineering Research, 13(11), 9758–9767.

Hanna, M., Elsayed, A., & Mostafa, M.-S. M. (2018). Automated software testing frameworks: A review. International Journal of Computer Applications, 179(46), 22–28. https://doi.org/10.5120/ijca2018917171

Jenkins. (n.d.). Retrieved February 14, 2022, from https://www.jenkins.io/

Kim, D. E., & Park, J. (2007). Application of adaptive control to the fluctuation of engine speed at idle. Information Sciences, 177(16), 3341–3355. https://doi.org/10.1016/j.ins.2006.12.021

Kim, E. H., Na, J. C., & Ryoo, S. M. (2009). Implementing an effective test automation framework. Proceedings of the International Computer Software and Applications Conference, 2, 534–538. https://doi.org/10.1109/COMPSAC.2009.188

Kumar, D., & Mishra, K. K. (2016). The impacts of test automation on software's cost, quality and time to market. Procedia Computer Science, 79, 8–15. https://doi.org/10.1016/j.procs.2016.03.003

Kuusinen, K., Gregory, P., Sharp, H., Barroca, L., Taylor, K., & Wood, L. (2017). Adopting test automation on agile development. https://doi.org/10.1007/978-3-319-57633-6

Mahajan, S., & Halfond, W. G. J. (2015). WebSee: A tool for debugging HTML presentation failures. Proceedings of the 2015 IEEE International Conference on Software Testing, Verification and Validation. https://doi.org/10.1109/ICST.2015.7102638

Mishra, A., & Atesogullari, D. (2020). Automation testing tools: A comparative view. International Journal on Information and Computer Security, 12, 63–76.

Nass, M., Alégroth, E., & Feldt, R. (2021). Why many challenges with GUI test automation (will) remain. Information and Software Technology, 138, 106625. https://doi.org/10.1016/J.INFSOF.2021.106625

Pajunen, T., Takala, T., & Katara, M. (2011). Model-based testing with a general-purpose keyword-driven test automation framework. Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops, 242–251. https://doi.org/10.1109/ICSTW.2011.39

Pelivani, E., & Cico, B. (2021). A comparative study of automation testing tools for web applications. Proceedings of the 10th Mediterranean Conference on Embedded Computing, 7–10. https://doi.org/10.1109/MECO52532.2021.9460242

Pradhan, L. (2011). User interface test automation and its challenges in an industrial scenario.

Rafi, D. M., Moses, K. R. K., Petersen, K., & Mäntylä, M. V. (2012). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. Proceedings of the 7th International Workshop on Automation of Software Test, 36–42. https://doi.org/10.1109/IWAST.2012.6228988

Ramler, R., & Wolfmaier, K. (2006). Economic perspectives in test automation. https://doi.org/10.1145/1138929.1138946

Robot Framework. (n.d.). Retrieved June 22, 2021, from https://robotframework.org/

SeleniumHQ. (n.d.). Selenium browser automation. Retrieved June 22, 2021, from https://www.selenium.dev/

TestNG. (n.d.). Retrieved June 22, 2021, from https://testng.org/doc/

Thummalapenta, S., Sinha, S., Singhania, N., & Chandra, S. (2012). Automating test automation. Proceedings of the International Conference on Software Engineering, 881–891. https://doi.org/10.1109/ICSE.2012.6227131

Vila, E., Novakova, G., & Todorova, D. (2017). Automation testing framework for web applications with Selenium WebDriver: Opportunities and threats. ACM International Conference Proceeding Series, 144–150. https://doi.org/10.1145/3133264.3133300

Wang, F., & Du, W. (2012). A test automation framework based on web. Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science, 683–687. https://doi.org/10.1109/ICIS.2012.21

Wiklund, K., Eldh, S., Sundmark, D., & Lundqvist, K. (2017). Impediments for software test automation: A systematic literature review. https://doi.org/10.1002/stvr.1639