TPM Vol. 22, No. 3, 2015
ISSN: 1972-6325
https://www.tpmap.org/

Open Access

IMPROVING SOFTWARE QUALITY AND TESTING EFFICIENCY THROUGH AUTOMATION FRAMEWORKS AND PROCESS OPTIMIZATION

BRINDA PRADEEP

LEAD ENGINEER HCL TECHNOLOGIES LIMITED.

Abstract
The new enterprise applications necessitate scalable testing methods that can accommodate frequent releases without sacrificing quality and reliability of software. This article gives the design and implementation of automated testing solutions of enterprise business applications with emphasis to functional, regression, compatibility and build verification testing. The project involved creating automated test suites with Selenium WebDriver, developing automation tools that would improve the productivity, and introducing process improvements that would decrease the number of manual efforts and boost the test cycles. Besides the execution of complex business processes using automated systems, an in-house automation utility was designed to simplify the routine testing processes, and hence the savings in costs and effort saved were quantifiable. The article shows how test automation and optimization of the engineering processes can be strategically adopted to enhance the quality of software, to ensure greater coverage of tests and also to efficiently deliver enterprise software systems.

Keywords: Automation Framework, Process Optimization, Selenium, Enterprise, Software system.

1. Introduction

Modern-day enterprise software systems are massive and complicated. They have numerous modules that are interlinked and require regular updates to ensure that they are in the market. Development teams are pressed towards real-time delivery of new features and at the same time ensure that the software remains stable and does not crash. This stress puts great burden on the process of testing, as the process of testing can be considered one of the most time consuming in software development process. Testing has the tendency of taking nearly half of all resources that are incurred in a software project (Devasena & Valarmathi, 2012). This cost increases even more when there is a manual testing, as manual test execution will also take more time and will also be subject to human error.

Research in the industry has discovered that the overall testing processes are typically undertaken using a systematic procedure although application of automation is still hampered by constraints of usability and tools (Kasurinen et al., 2010). Most organizations still use manual functional check, manual regression cycle, and manual compatibility check, although all these activities are monotonous and predictable. Availability of compatibility testing is rather a challenge since enterprise systems are used on numerous combinations of browsers, devices and operating environments and it is impractical to test all combinations (Yoon et al., 2008). With shorter release cycles, there is a increasingly greater disparity between what must be tested and what can be effectively tested manually.

The current paper presents a report on a real-life project to bridge that gap with automation structures and optimization of the processes. The work discusses four testing areas that are usually prevalent in enterprise applications. They include functional testing, regression testing, compatibility testing and build verification testing. Selenium WebDriver allowed us to create automation test suites to perform these testing activities, with minimal manual intervention. Industry-specific automation utility was also developed to help in eliminating manual repetitive steps which could not be eliminated through the use of automated scripts. The framework layout, approach to implementation, and quantified outcomes, with respect to runtime, defect detection, test-coverage, and cost-savings, are described in the paper. The rest of the sections are arranged in the following way. Section 2 looks at related work on test automation. Section 3 outlines the methodology and design of the framework. Section 4 describes how we will implement the automated test suites and other un-supporting tools. Results and discussion are given in section 5. Section 6 gives a conclusion to the paper.

2. RELATED WORK

The automation of tests has been addressed in numerous ways, that start as low as unit level automation tools and go up to the full system level automation strategies. Zamli and Isa proposed an automated unit testing tool implemented in Java which minimized human input in creating test data and able to execute test data in parallel (Zamli & Isa, 2008). They demonstrated that testing cost can be effectively reduced by automating not only test-execution, but repetitive preparation steps, as well. Strecker and Memon (UI level) investigated the factors that actually affect fault detection in testing and discovered that statement coverage and GUI event coverage were closely linked with the effective fault detection (Strecker & Memon, 2008). This observation confirms the notion that coverage measures are the right indicator of test design automation, as opposed to convenience.

The other path to automation lies in model-based testing. In an effort to unify all the varieties of approaches to model-based testing under one taxonomy, Utting, Pretschner, and Legeard helped teams adopt an approach according to the attributes of their systems (Utting et al., 2011). Similar interest has been given to regression testing. Jin, Orso, and Xie came up with a behavioral regression testing technique that would directly compare two versions of a program, instead of simply relying on a set of test cases which was capable of identifying actual regression errors in a pre-test assessment (Jin et al., 2010). Karhu, Repo, Taipale, and Smolander looked at why certain organizations practice automation commonly and others do not and found out that the products which are generic, stable and are reused across projects were the ones which were most likely to gain value through automation (Karhu et al., 2009). The results of their work apply to enterprise applications, as most enterprise modules, including login, reporting and notification processes, are consistent and repeated between releases.

Itkonen, Mantyla and Lassenius examined the manual testing experiences at multiple companies given and found that there is a great variety of strategies and techniques deployed by testers without a distinct design phase (Itkonen et al., 2009). This demonstrates that manual testing skill and automation are not opposites, but can be used together as testers usually include judgment that cannot be entirely brought by scripted automation. These studies imply that an efficient automation program must have a clear framework, coverage conscious test-design and understanding of the areas of manual judgment value addition. Figure 1 displays the structure of layered automation framework that is adopted in this work that lies between the layers of presentation, business, and data of the enterprise application.

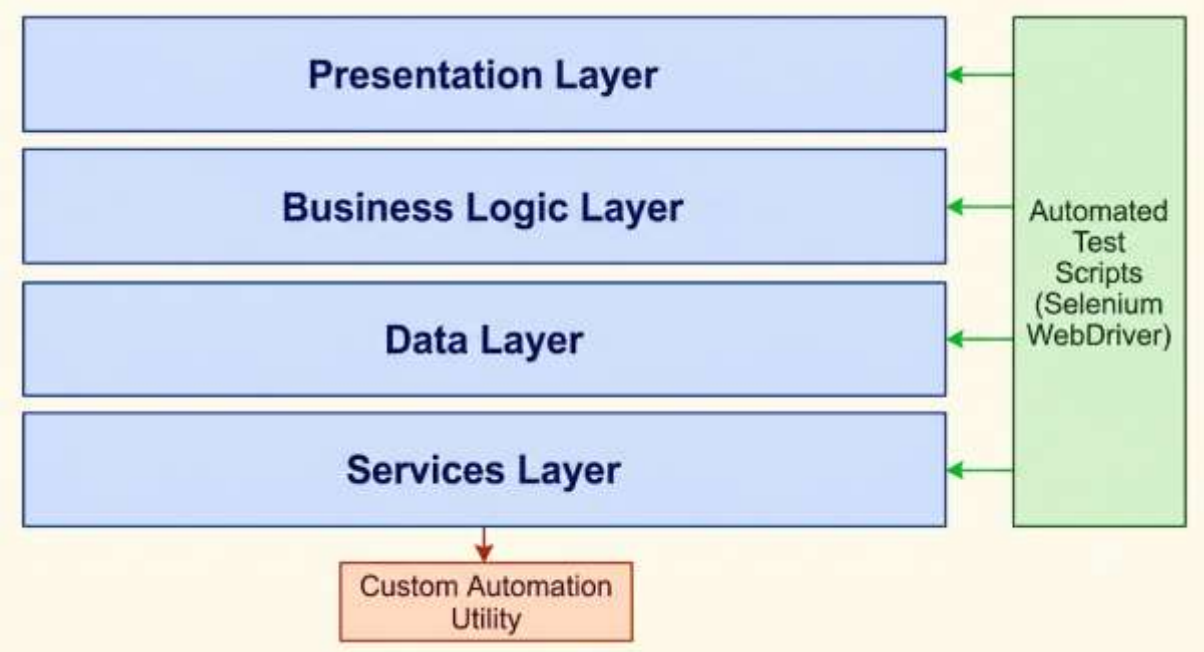


Figure 1. Layered architecture diagram of presentation, business logic, data and services layers of the automation framework and the interaction of the automated test scripts with each layer.

3. FRAMEWORK AND METHODOLOGY

The automation structure was constructed based on a layered architecture like the n-tiered automation architecture developed by Day which isolates the issues of presentation, business logic, data and services into service, data, business and presentation concerns (Day, 2014). The rationale behind this separation is that enterprise applications are frequently changed at the presentation layer such as with a redesign of the user interface and that the underlying business rules are less frequently changed. The framework offered the benefit of isolating the presentation level automation scripts to the business logic scripts to minimise rework required when an interface was changed alone.

Selenium WebDriver was chosen as the base automation framework due to its ability to interact with the browser on the mainstream browsers in enterprise settings and its ability to interact with and integrate with popular programming languages practiced by the testing team.
The paradigm was applied to four types of testing. Functional testing was done to ensure that specific features performed as per specified business needs. Regression testing was used to ensure that there was no breakage of existing functionality by new code changes. Compatibility testing was done to ensure that the application was compatible with expected browser and operating system combinations. Build verification testing had a quick check following each new build to ensure core functions started up to make sure substantial testing commenced. The manual testing effort should be gathered and documented at the baseline of the automation before the automation process, in the manner of observational studies of manual testing practice, where actual tester practice is monitored instead of just projected by documentation alone (Itkonen et al., 2009).
Selection criteria of automation candidates included their frequency of execution, stability of its underlying feature and business criticality. The features that were implemented in each release cycle, and whose structural implementation was not going to change, were first implemented in an automated way. This choice rationale minimized the likelihood of committing automation resources to scripts that would soon-to-be rendered obsolete. The parameterization of functional and regression script test data meant that they could be run with a multiplicity of input combinations, and they provided more coverage, without the number of individual scripts being increased. The automated test scripts were being managed by version control, and a review went through with each script before being integrated into the main automation suit.

4. IMPLEMENTATION OF AUTOMATED TEST SUITES

The functional and regression suites were the initial ones to be implemented, as they constituted the biggest proportion of manual repetitious activity. The application was met by Selenium WebDriver scripts that were used to engage with the application based on the user interface to simulate actual user activities like form submission, navigation and data validation. The compatibility test scripts were rewritten to be compatible with many different browser settings based on a common script base, which did not need to implement the same logic in each browser. Build verification scripts were deliberately small and only contained critical paths so that they could run fast after each new build without slacking development pipeline.
In conjunction with script development, a dedicated auto utility was developed to perform repetitive tasks, which were not considered a part of the general Selenium suites, including environment setup, seeds test data and post-test run logs. This utility lowered the amount of time that it took to prepare manually before every test cycle was completed. This strategy follows a wider strategic trend in the industry where teams are deploying visual and interaction-based automation to substitute manual system level tests often of tedious nature as shown in a case study at an industrial level where visual GUI based testing had beneficial impacts on running time and bug detectability following initial transition difficulties and challenges (Alegroth et al., 2013). In advance of full-scale automation, a lightweight analysis of cost and benefit was conducted that resembles simulation-based decision support models where teams may decide to determine whether automation investment will be cost effective to a selected as a given testing activity (Sahaf et al., 2014).
Table 1 shows the time that was measured on the execution of each type of testing prior to the introduction of automation and after its application. This is compared on the average time for each category to complete one complete test cycle.

Table 1. Test suites were compared on the average time in hours that the tests were run before and after the test suites were automated.

Testing Category	Manual Execution Time (hours)	Automated Execution Time (hours)	Time Reduction (%)
Functional Testing	40	12	70
Regression Testing	60	15	75
Compatibility Testing	50	18	64
Build Verification Testing	20	4	80

Table 1 results indicate that the highest percentage reduction was found on the build verification testing, as they were straightforward and repetitive and thus were appropriate to be automated. Strong improvement is also logical, as regression suites are run quite often, in numerous releases. The trend in total execution time of the five consecutive release cycles of manual and automated execution is provided in Figure 2.

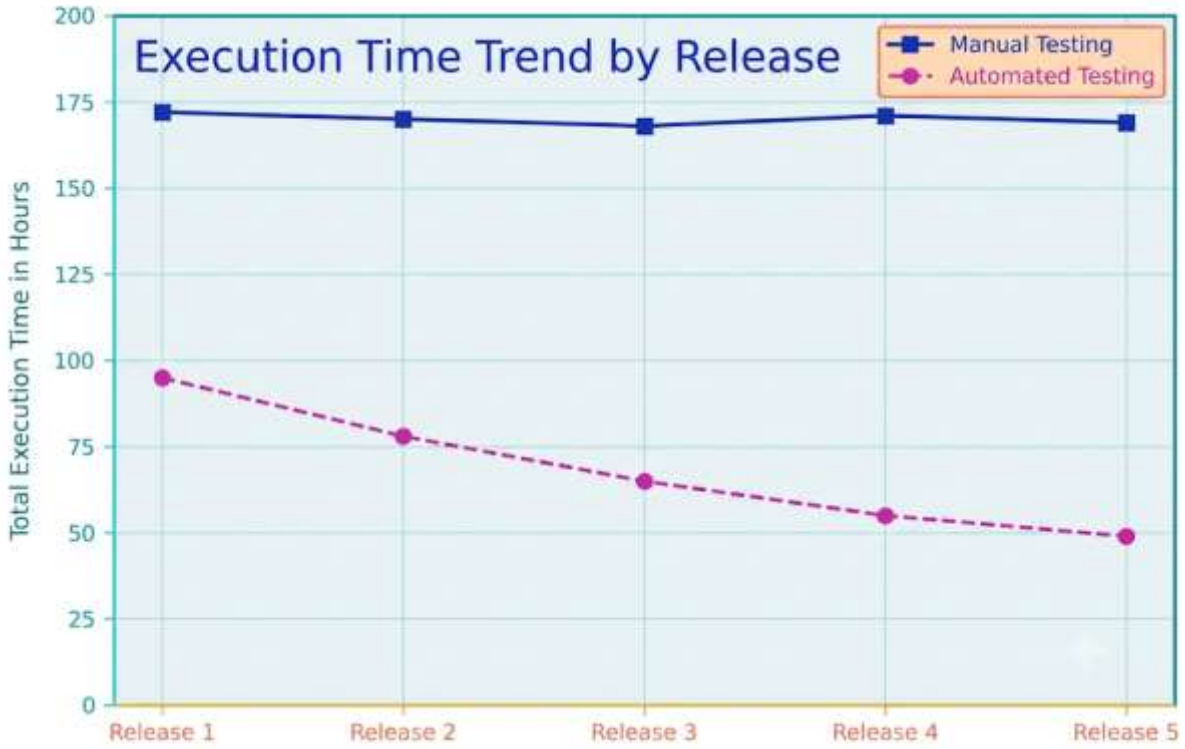


Figure 2. Comparison of total time of test execution of five release cycles of automated test and manual test in hours using a line graph.

Parallel tracking on defect detection and execution time was used to ensure faster testing was not done at the expense of quality. Table 2 indicates how many defects were found each release cycle prior to and after automation.

Table 2. Defects per release cycle found with manual methods of testing versus automated methods of testing in five releases.

Release Cycle	Defects Detected (Manual)	Defects Detected (Automated)
Release 1	45	52
Release 2	40	58
Release 3	38	61
Release 4	35	65
Release 5	33	70

The trend in Table 2 reveals that automated testing has been able to identify more defects in each release cycle and that this disparity has increased as the automated test suite has matured and acquired additional cases with time.

5. Results and Discussion

In addition to the execution time and defect rates, test coverage was also measured at the module level to be able to see how well each aspect of the enterprise application was being tested. Table 3 shows the pre and post automation initiative percentages of five core modules.

Table 3. Percentage of test coverage achieved with five core modules of application on how tests were run prior to the automated test suites implementation and after their implementation.

Module	Coverage Before Automation (%)	Coverage After Automation (%)
Ticketing	55	88
Payment	60	92
User Management	58	85
Reporting	50	80
Notification	45	78

This has increased coverage as presented in Figure 3 in bar chart format, which can be compared in all the five modules.

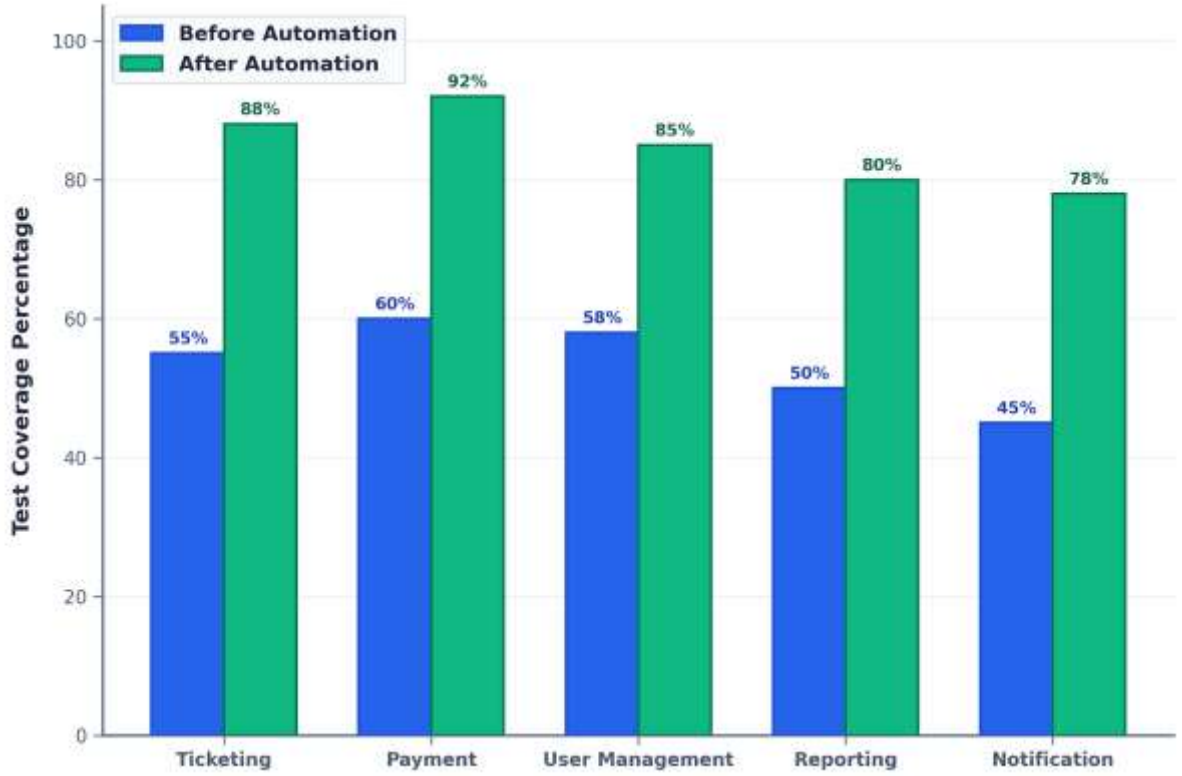


Figure 3. Comparison of percentage of test coverage in five application modules both before and after application of automation.

The coverage growth helps to support the problem that with the automation; the teams will be able to test more parts of the application without correspondingly increasing their efforts. This is in line with survey results about regression testing where it is observed that as more tests are added to the test suites, there is a need to carefully select and prioritize test cases in order to ensure that the test execution is feasible but still optimizes on the detection of faults (Yoo & Harman, 2010). In this program, parameterized test scripts enabled a greater number of combinations of inputs to be run within the same execution period, which is one of the reasons why there was coverage improvement as seen in Table 3.
The reductions in efforts in Table 1 have been converted into monetary calculations to determine the cost savings, with the help of the usual internal cost rates related to tester hours. These findings are summarized in table 4.

Table 4. Cost and time saved per type of testing, using values of manual and automated hours and cost/hours of testing.

Testing Category	Effort Saved (hours per cycle)	Effort Reduction (%)	Estimated Cost Saved (USD per cycle)
Functional Testing	28	70	2800
Regression Testing	45	75	4500
Compatibility Testing	32	64	3200
Build Verification Testing	16	80	1600
Total	121	71	12100

Figure 4 presents a cumulative cost savings in 6 months of automation implementation, where the level of savings increased with additional automated scripts to the suite.

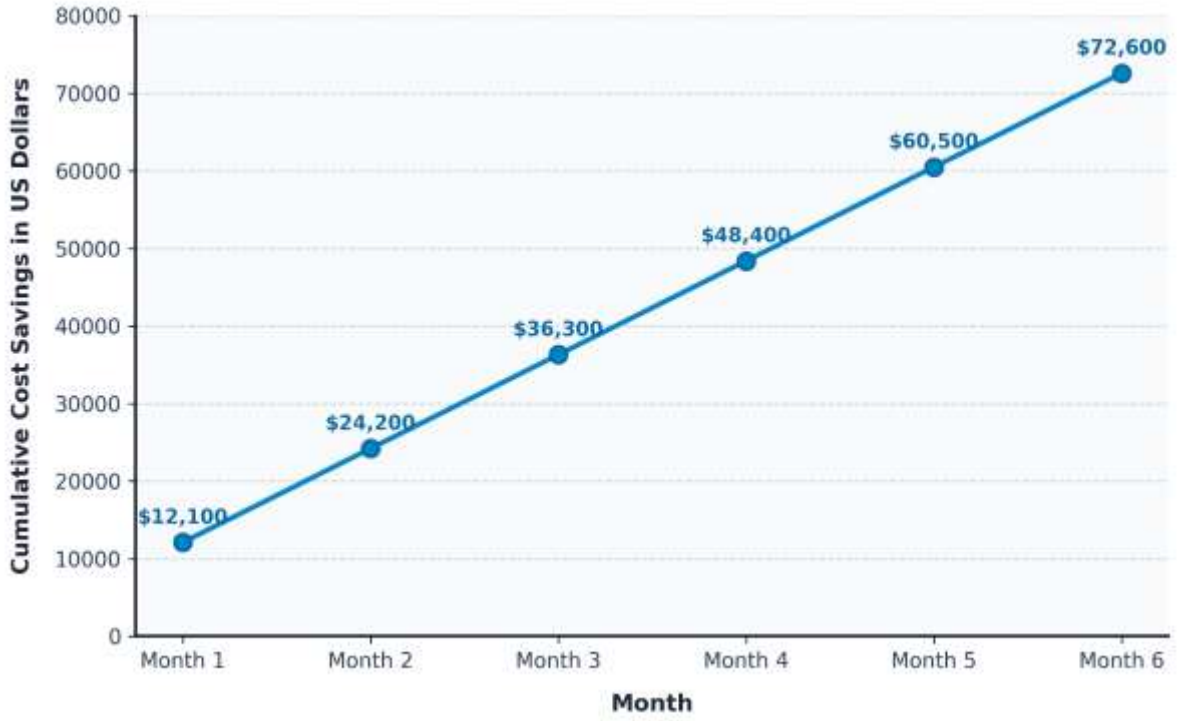


Figure 4. Trend diagram illustrating the cumulative savings in US dollars during six months of implementation of the automated testing system.

Although the quantitative results were favorable, the qualitative comments that testers made showed that there were some issues that cannot be enumerated by numbers. This is in accordance with the previous studies that found out that even after new processes are introduced, the redundant and overlapping testing effort may still exist especially due to the lack of clarity in spelling out responsibility between test levels (Engström & Runeson, 2012). The key themes according to testers after automation rollout were summarized by way of Table 5, which was based on structured feedback.

Table 5. Compilation of qualitative ideas that were expressed by testers in the process of negative feedback on the implementation of the automation framework.

Theme	Tester Observation
Learning curve	To learn how to write and maintain Selenium WebDriver scripts, testers required to have some initial training time to become at ease.
Script maintenance	User interface changes incurred new overhead costs in scripts because it needed to adjust with every change, and this had to be included in the testing team.
Confidence in coverage	Users of the regression model were reported to be more confident in the results as more test was being put in a wider and consistent manner.
Tool reliability	There was a little instability in the automation environment observed in the first month of deployment.
Team collaboration	Developers and testers had to be closer coordinated to ensure that automated scripts were in sync with changes in the code.

The maintenance issues discussed by the testers represent a recognized challenge in test automation, the continuing expenses of script maintenance can detract the time saved initially unless controlled well. To deal with such an issue, structured improvement models have been suggested that establish measurable key areas to evaluate and continuously enhance test code quality through time that could be used to refine the automation suite in the future (Eldh et al., 2014). This burden was minimized by the custom automation utility, which centralized the logic of setting up environments and data outside individual test scripts, so that the scope of a single change to the interface is more limited.

The findings indicate that automation also decreased the time of execution, increased the number of defects, increased coverage, and generated cost savings that could be measured in addition to creating new tasks such as script maintenance and coordinate groups. The findings are generally in line with the broader body of research

regarding the adoption of test automation, which is uniformly reported to have led to efficiency gains as well as apparent difficulties to successful adoption amongst various organizational settings.

6. CONCLUSION

This paper introduced the design, implementation, and testing of an automation structure in an attempt to enhance the quality of software and efficiency of software testing in regards to enterprise business applications. Selenium WebDriver was used to construct automated test suites that were applied across functional, regression, compatibility, and build verification testing and were backed by a custom automation utility that lessened the tedious manual preparation efforts. The layered framework design decoupled the presentation, business logic and data concerns and this reduced the effect that interface modifications had on the automated scripts. There were measured improvements in execution time (between 64 percent and 80 percent in testing categories) and defect detection (better) and test coverage (greater) in core application modules. The cost analysis indicated a cumulative savings of 71 percent and projection of 12100 US dollars of full test cycle saved cost. Qualitative feedback indicated persisting problems related to script maintenance and tool untrustworthiness, which hints at the necessity of meticulous measurement and enhancement of the automation package in the long run. The results confirm the observation that a properly formulated automation structure and a specific process optimization can significantly enhance the speed and reliability of enterprise-wide testing of software and also indicate that working on automation successfully cannot be achieved through a single-time effort to address maintenance practice and collaboration among its team members.

REFERENCES

1. Alegroth, E., Feldt, R., & Olsson, H. H. (2013). Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study. Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study. https://doi.org/10.1109/icst.2013.14
2. Day, P. (2014). N-Tiered Test Automation Architecture for agile software systems. Procedia Computer Science, 28, 332–339. https://doi.org/10.1016/j.procs.2014.03.041
3. Devasena, G., & Valarmathi, M. (2012). Multi agent based framework for structural and model based test case generation. Procedia Engineering, 38, 3840–3845. https://doi.org/10.1016/j.proeng.2012.06.440
4. Eldh, S., Andersson, K., Ermedahl, A., & Wiklund, K. (2014). Towards a Test Automation Improvement Model (TAIM). Towards a Test Automation Improvement Model (TAIM), a, 337–342. https://doi.org/10.1109/icstw.2014.38
5. Engström, E., & Runeson, P. (2012). Test overlay in an emerging software product line – An industrial case study. Information and Software Technology, 55(3), 581–594. https://doi.org/10.1016/j.infsof.2012.04.009
6. Itkonen, J., Mantyla, M. V., & Lassenius, C. (2009). How do testers do it? An exploratory study on manual testing practices. How Do Testers Do It? An Exploratory Study on Manual Testing Practices, 494–497. https://doi.org/10.1109/esem.2009.5314240
7. Jin, W., Orso, A., & Xie, T. (2010). Automated Behavioral Regression Testing. Automated Behavioral Regression Testing, 137–146. https://doi.org/10.1109/icst.2010.64
8. Karhu, K., Repo, T., Taipale, O., & Smolander, K. (2009). Empirical Observations on Software Testing Automation. Empirical Observations on Software Testing Automation, 201–209. https://doi.org/10.1109/icst.2009.16
9. Kasurinen, J., Taipale, O., & Smolander, K. (2010). Software test Automation in Practice: Empirical observations. Advances in Software Engineering, 2010, 1–18. https://doi.org/10.1155/2010/620836
10. Sahaf, Z., Garousi, V., Pfahl, D., Irving, R., & Amannejad, Y. (2014). When to automate software testing? decision support based on system dynamics: an industrial case study. When to Automate Software Testing? Decision Support Based on System Dynamics: An Industrial Case Study, 149–158. https://doi.org/10.1145/2600821.2600832
11. Strecker, J., & Memon, A. (2008). Relationships between Test Suites, Faults, and Fault Detection in GUI Testing. Relationships Between Test Suites, Faults, and Fault Detection in GUI Testing, 12–21. https://doi.org/10.1109/icst.2008.26
12. Utting, M., Pretschner, A., & Legeard, B. (2011). A taxonomy of model-based testing approaches. Software Testing Verification and Reliability, 22(5), 297–312. https://doi.org/10.1002/stvr.456
13. Yoo, S., & Harman, M. (2010). Regression testing minimization, selection and prioritization: a survey. Software Testing Verification and Reliability, 22(2), 67–120. https://doi.org/10.1002/stvr.430
14. Yoon, I., Sussman, A., Memon, A., & Porter, A. (2008). Effective and scalable software compatibility testing. Effective and Scalable Software Compatibility Testing, 63–74. https://doi.org/10.1145/1390630.1390640
15. Zamli, K. Z., & Isa, N. a. M. (2008). JTST – an automated unit testing tool for Java program. American Journal of Applied Sciences, 5(2), 77–82. https://doi.org/10.3844/ajassp.2008.77.82