Overcoming Test Debt and Advancing Software Sustainability with Automated Testing: A B2B Trading Platform Case Study

Xiaoge Zhang
Technical University of Darmstadt
liriaxiaoge@gmail.com

Bhavika Sharma
Technical University of Darmstadt
bhavika.sharma@stud.tu-darmstadt.de

Timo Koppe
Technical University of Darmstadt
timo.koppe@tu-darmstadt.de

Abstract

This paper explores the current state of software quality assurance (SQA) practices with a focus on test automation. It applies these practices to a B2B online trading platform and builds a conceptual framework to implement automated end-to-end (E2E) tests for a critical business process and develop a test debt payback approach to increase unit test coverage. Our research follows the Design Science Research (DSR) methodology and offers deep insights through collaboration with the business and development team. The paper concludes by providing strategical and operational recommendations for organizations looking to improve their SQA processes. Overall, the study highlights the importance of SQA and test automation for long-lasting software and demonstrates concrete approaches for solving common challenges in the field.

Keywords: E2E Testing, Test Automation, Sustainable Software, Test Debt Payback, Case Study

1. Introduction

Software is ubiquitous. With the rapid development of information technology and software engineering in the last decades, software has become a central part of our society. The consequences of insufficient software quality stretch from minor inconveniences in daily life to cyberattacks against government agencies (e.g. Walkinshaw, 2017). From a business perspective, software quality assurance is not only vital for the success of software development projects but also a major cost driver due to the high complexity and dynamics of today's software systems. However, numerous research has also shown the return (Slaughter et al., 1998) and cost savings (Tassey, 2002) that early investment in software quality could provide. Despite the universal acknowledgement of its importance, software quality has no canonical definition as it depends greatly on the rapidly shifting IT landscape and business requirements. Several software quality models have been proposed by researchers in the software engineering field in attempt to formalize software quality. As an example, the latest PAS-754 specifies safety, reliability, availability, resilience, and security as five trustworthiness aspects that concern software operation (Walkinshaw, 2017).

Software inspection, measurement and testing are the major instruments of software quality assurance. While inspection activities such as code review examine the software statically and measurement controls software quality through quantitative metrics and indices, software testing executes the system under test (SUT) with a primary intent of discovering software failures and system validation. Through proper but simple unit testing, a great majority of production failure could already be caught (Yuan et al., 2014). Using automation tools that reduce the manual generation, execution and analysis of tests, test automation could further improve the efficiency and ROI of software quality assurance. However, testing of complex software systems under agile development processes faces numerous challenges. Rapid development cycles make verification and validation the first to be sacrificed in case of resource shortages (Torkar and Mankefors, 2003), and testing budgets are frequently exceeded in practice (Ng et al., 2004). Furthermore, environmental factors such as shorter product lifecycles and rapid development of new IT-based business models in recent years impose additional cost and performance pressure on IT departments (Jamil et al., 2016).

URI: https://hdl.handle.net/10125/107318
978-0-9981331-7-1

In response to these challenges, a variety of practices, tools and approaches have been developed in the field of software testing (e.g., Ateşoğulları and Mishra, 2020). Organizations must conduct significant research, evaluation, and decision-making effort to answer the questions of when and what to automate (Garousi and Mäntylä, 2016). Establishing an efficient testing process is hence a non-trivial and context-dependent exercise. By illustrating the common challenges of quality assurance through the example of a real-world B2B online trading platform, this paper aims to illustrate an approach for designing quality assurance for modern web applications with a focus on testing and test automation. Conducting a complete design cycle, we worked closely with the development and management team of the B2B online trading platform, observed and analyzed the challenges faced by the organization while establishing test automation, and derived transferable knowledge for future research and implementations.
The rest of the paper is organized as the recommended publication schema for design science research (Gregor and Hevner, 2013). Section 2 presents the theories and practices of software quality assurance and software testing. Section 3 describes the applied research methodology and section 4 presents the designed artefacts as well as their evaluations. Subsequently, section 5 interprets the evaluation results and discusses learnings of the design process. In the end, section 6 concludes the paper.

2. Theoretical Background

In this section, we delve into the theoretical foundations of Software Quality Assurance (SQA), Software Testing, and Test Automation, exploring their essential roles in achieving high-quality software products and efficient development processes.

2.1. Software Quality Assurance

SQA can be understood as part of software quality management, separate from software quality control and testing. It focuses on organizing and controlling the software development process rather than testing against requirements. SQA aims to ensure a certain level of confidence in software quality, covering various stages from requirement engineering to defect monitoring and resolution. The Capability Maturity Model for Software (CMM) (Humphrey, 1988) justifies the need for SQA in organizations. CMM assesses the maturity of an organization's software development processes and highlights the importance of quality software delivery, customer satisfaction, and process improvement. The model describes five maturity levels: Initial, Repeatable, Defined, Managed, and Optimizing. In order to increase its process maturity level, an organization must implement the necessary key practices of the higher maturity level. SQA is a key practice at the Repeatable level, which involves planning SQA activities, adherence to standards and procedures, informing stakeholders, and resolving noncompliance issues (Paulk et al., 1993). In order to help organizations establish SQA, the International Organization for Standardization (ISO) has developed the ISO 90003 guideline (ISO/IEC/IEEE, 2018) for applying the ISO 9001 standard to software quality assurance. ISO 90003 provides recommendations for test planning, addressing test types, objectives, cases, and data, as well as documenting validation results (ISO/IEC/IEEE, 2018). ISO 90003 registration is pursued by software companies for international recognition and competitive advantage (Helio Yang, 2001). Total Quality Management (TQM) is another approach to software quality, promoting continuous improvement through humanistic principles and scientific methodologies (Parzinger and Nath, 2000). TQM implementation requires significant cultural changes and encompasses factors such as executive commitment, quantitative quality control, process evaluation, and training in various areas, including statistical methods and ISO 9000 principles. TQM adoption has shown positive impacts on customer satisfaction, CMM levels, and ISO 90003 compliance (Parzinger and Nath, 2000).

2.2. Software Testing

Software testing is the process of executing a program under specific conditions and evaluating its behavior (IEEE, 2014). It aims to determine if the program functions as expected. The testing process can be categorized as unit testing, integration testing, or system testing, depending on the scope of the system under test (SUT).

Test-Oriented Development. With the emergence of agile software development, various test-oriented DevOps methodologies have been proposed to ensure efficient and effective testing processes. These methodologies provide rules and guidelines for writing new features and their corresponding tests. Traditionally, developers would write tests after completing the production code for a unit or system. However, these methodologies introduce a different workflow. Following are three agile approaches that describe the structuring of the test suite:

1. Test-driven development (TDD): TDD involves short coding cycles that begin with writing a unit

test, followed by writing the production code, and concluding with code refactoring. Each cycle should be completed within 10 minutes and at a steady rhythm. TDD improves functional quality and productivity due to its granular and uniform coding cycles compared to the traditional waterfall testing approach (Fucci et al., 2017).

2. Acceptance test-driven development (ATDD): ATDD is a variation of TDD where the development process is driven by acceptance tests instead of unit tests. Developers generate test cases from the requirements to verify the functionality of the SUT. TDD and ATDD share the challenge of high dependency between test code and system implementation, as code writing and test case generation occur in the same coding cycles (Solis and Wang, 2011).

3. Behavior-driven development (BDD): BDD addresses the challenges of ATDD by focusing on test automation. Main characteristics of BDD are the use of ubiquitous language in writing specification, iterative processes that decompose business outcomes into system features, and automated acceptance testing (Solis and Wang, 2011).

2.3. Test Automation

Test automation involves the use of automation tools and frameworks to automate the activities of test generation, execution, and evaluation and is expected to improve accuracy of fault detection, increase test coverage, and improve overall efficiency of software testing (Umar and Chen, 2019). Through a multi-vocal literature review, (Garousi and Mäntylä, 2016) proposed a checklist to support the decision-making on when to automate software testing. Frequent regression testing, importance of tests, economic benefits, and stability of test interface were a few key factors favouring automation. Factors against automation were major modifications of SUT, complexity and dependence on other products, indeterministic test execution, and maintenance effort. Upon deciding for test automation, developers and testers face the next challenge of automating existing manual test cases and maintaining the automated test suite over the time, which requires immense effort. Test automation frameworks are systems that provide core functionalities for creating, executing, and maintaining test cases, and allow extension with new kinds of tests. These fulfill high-level requirements of large-scale test automation: automatic test execution, ease of use, and maintainability. Moreover, test automation frameworks should also be capable of error and failure handling, test results verification, detailed logging, and automatic reporting (Laukkanen, 2006). Beside using an appropriate framework, test automation also faces organizational challenges. Through a systematic literature review, (Wiklund et al., 2017) constructed a socio-technical system of test automation. Behavioral factors, namely process adherence and organizational change affect the deployment and success of test automation projects. Problems in business and planning such as lack of time and resources often lead to accumulation of test debts, i.e., inadequate test coverage and improper test design. It is hence important to take both technical and organizational factors into consideration when planning and implementing test automation.

3. Research Method and Case Description

The purpose of this study was to develop and design quality assurance for a real-world web application with a focus on test automation and derive transferable knowledge from the process. Design is both a process and a product, and design science is one of the two paradigms that characterize the Information Systems research (Hevner et al., 2004). Design science seeks to solve real-world problems and generate design knowledge to help practitioners create solutions to problems in specific fields (Engström et al., 2020). In this study, we follow the Design Science Research Process (DSRP) described by Pfeffers et al. (2020), a process sequence beginning with identification of a problem followed by the iterative activities of objective-setting, design and development, demonstration, evaluation, and communication (Peffers et al., 2020). The output of DSRP in IS context are IT artefacts that can be implementations or practices such as an automated test suite and a set of quality assurance processes, as is the case in this study. The real-world problem of this study occurs at a B2B online trading platform of a German technology group. The platform is responsible for real-time displaying and trading of commodities, processing of the transactions, as well as management of client accounts. Billions of euros of revenue are generated through the trading platform. Due to the high complexity and associated financial risks, the functional correctness of the application is of particular importance. The trading component of the application is the most critical amongst all, having the biggest financial impact in case of serious defects and is hence selected as the object of this study. The current test suite for the application is

mainly made of unit tests while integration tests are missing. Test scope focuses on the aforementioned trading component, and unit test coverage is very low. Due to the lack of a complete automated test suite, regression testing is conducted completely manually by the domain experts. The team is provided with the continuous test automation tool Tricentis Tosca[1], which has demonstrated success in the company's SAP testing. For the SUT only a few UI tests have been implemented in Tosca without fully exploiting the tool's full capability. Test cases are designed by the domain experts and documented in Excel spreadsheets. The team works on the development of the SUT in 4-week Scrum-Sprints and has a release frequency of 1-2 months. Despite the long release cycle, test debt repay has been slow and the current testing is considered far from ideal by both the managers and the developers. In response to these challenges, the management wished to reduce long-term risks by improving the overall testing of the SUT and introducing best practices to the team. Specifically, they aimed to apply and evaluate Tricentis Tosca for End-to-End test automation; revise and improve current quality assurance processes; and derive transferable knowledge for further development of the SUT and other applications of the organization. From a research perspective, this study aimed to investigate the gap between the practical implementations and the state-of-the-art theories of quality assurance with a focus on automated software testing. This included developing artefacts to satisfy stakeholders' requirements and identifying causes for the current organizational problems. The developed artefacts were evaluated quantitatively as well as qualitatively through interviews and surveys. Adhering to the Design Science Research guidelines (Hevner et al., 2004), the artefacts were communicated to stakeholders in the forms of presentations, document reports, meetings, and final demonstrations of results.

4. Results

In this section we present our three main results. First, we discuss the framework we provide to understand the problems of test automation embedded in organizational and cultural factors and provide countermeasures to address these. Second, we show the additional value of deploying test automation leading to easier software maintenance by increasing test coverage and ROI. Thirdly, we design a test debt payback approach which aims to structure and organize testing related tasks alongside feature development and operational tasks.

[1]https://www.tricentis.com/products/automate-continuous-testing-tosca

Problem Analysis Framework A design cycle in design science research starts with problem investigation to identify the areas for improvements (Wieringa, 2014). As described in the last section, the main problem of the organization was the lack of test management and improper implementation of testing. This was reflected by the large amount of accumulated test debt in the SUT. In order to find out the reasons behind the problem, semi-structured interviews were conducted with domain experts and developers of the organization. The interviews focused on covering four topics: individual backgrounds, role and experience in the team, opinion on the current testing, and suggestions for areas of improvement. Based on the data gathered from the interviews, we present a conceptual framework of causes and solutions as a starting point of the design. The interviews were conducted with relevant stakeholders of the SUT, including internal employees as well as subcontracted developers. The statements of interviewees were coded and clustered into the following four classes of factors:

1. Technology-level factors: factors regarding the technology involved, including the tools used for testing and the SUT's architecture and codebase.

2. Actor-level factors: factors regarding the individual participants. The two main groups of actors in this case are the developers and the domain experts. Main independent variables identified are the individual's technical and domain-specific know-how, the responsibilities carried, and the time and resources that are available to the individual.

3. Organization-level factors: factors regarding the organization of the team. Planning of testing and quality assurance, coordination of testing and development processes, prioritization of testing and implementation tasks and controlling mechanisms used for ensuring process conformation and transparency are the main identified categories.

4. Environment-level factors: factors regarding the environment of the specific case. This includes enterprise culture, that is the enterprise-wide background of software quality assurance and testing, and business domain, which is the specific field of industry where the SUT and its stakeholders situate.

Among 66 collected statements from 13 interviewees, organization- and actor-level factors emerged as the most prominent classes. Technology and environment

were significantly smaller classes that each contributed about 10% of all the statements. From all types of interviewees, the developers and the product owners offered the majority of statements, while managers and testers offered less due to their smaller head counts. Low quality codebase was the most frequently mentioned and universally agreed upon factor amongst all, especially the developers. The current developers faced a lot of difficulties caused by the legacy code that was produced by the initial outsourcing agency. The unit tests were poorly written and are barely useful nowadays, according to three developers. The large amount of broken unit tests and incomprehensible code in the codebase led to difficulty in testing and development. Reasons for the low-quality initial development were, according to business stakeholders, cost minimization and underestimation of expenses at the beginning of the project. Lack of tool support and utilization of available tools also forced testing to be predominantly manual. Architectural complexity of the SUT was a smaller but also important factor contributing to difficult test debt payback and lacking documentation. Due to the monolithic architecture and complex add-in landscape of the SUT, constant upgrades and adaptations to external changes were required, producing extra overhead for the developers, and further reducing the available testing time. Lack of resources and time was the most prominent actor-level factor. Pressure of feature development and issue resolution, and hence, a general lack of time for testing was a common complaint of developers. There was also an uneven distribution of know-how and test-related responsibilities in the team. The organization lacked the know-how for implementation of test automation in the corresponding tool. Since testing was manually performed by the domain experts with the necessary domain knowledge, it became a bottleneck for an on-schedule release when the responsible person was not available. In general, ownership of quality was absent in the organization, meaning that not everyone held themselves responsible for the quality of the product and processes. A test specialist who planned, organized, and established testing was a vacant role in the team. Test plan, requirement specification, technical documentation, quality metrics were all absent despite being important assets of quality assurance. Testing had been systematically relegated by product owners and developers. Not only domain experts but also developers were not sure about the test and risk coverage when asked. Testing was not coordinated and carefully considered during architectural design. Hence, the product also lacked a stable and isolated environment for testing as well as a basic set of test configurations. Extending to the overall development and operations, the project faced numerous challenges that were not properly managed. Despite using Scrum terminologies such as “sprint planning” and “sprint review”, conformance to the Scrum methodology was low. User stories and tasks were often incompletely documented and discussed during sprint planning, and the team frequently underestimated development effort. The project lacked transparent leadership and management, although its complexity demanded high interdivisional coordination and cooperation across multiple locations. At an environment-level, the business domain of the application and the enterprise culture were not particularly favorable either. High complexity of the online-trading domain made it more difficult to have transparent and efficient communication between domain experts and developers. Although the technology group is widely diversified and relatively well-digitalized, its industrial background and the enterprise-wide absence of software development culture led to the lack of strategical awareness for software quality and testing. Upper management that controlled the project budget was not well-informed about the challenges faced by the development team due to the hierarchical organizational structure, making it difficult for the requirements and feedback of the employees to be addressed.

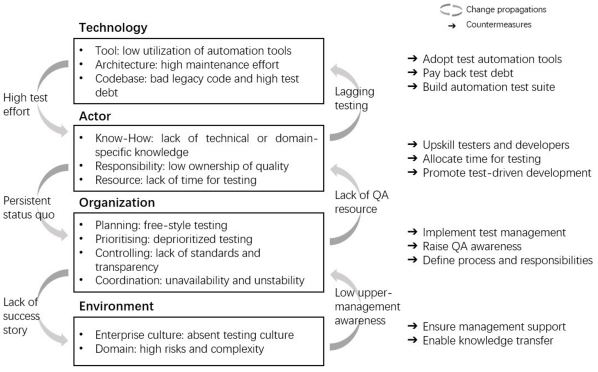

Figure 1. Conceptual framework: Impediments for test automation across organization layers and countermeasures

Value of Test Automation For the trading component of the SUT, test cases for an end-to-end (E2E) test scenario were implemented in Tosca. The test scenario encompasses the customer journey of a user on the trading platform that begins with log-in and ends with submission of a trading order. The validation of the test scenario concerns a dynamic price calculation based on specific price settings in the backend, which could

lead to critical financial damage if not done correctly. Manual testing of the test scenario was slow due to the complicated backend settings, large number of steps spanning three application interfaces, and requirements for multiple access rights. Different combinations of the test attributes of the SUT and their different values amounted to 54 possible test variants. Manual testing could not feasibly cover all these variants and during the time of the study only 2-3 variants were being tested. Manual testing of the test scenario had been performed regularly as a part of regression testing. The test steps were extensively documented with detailed explanations for the calculation. Three testers collaborated on the manual testing: one designed the Excel sheet based on business insights and intuition, one executed the test steps in the SUT and recorded the attributes in the Excel sheet, and one carried out the verification in the database. According to the testers, regression testing about 30 test cases of the test scenario typically spanned across multiple working days depending on the availability of the participants. With the help of documentations and explanations from the testers, an automated test suite of the test scenario was implemented in Tricentis Tosca. This automated test suite covered the entire testing process of test case design, test scripting, test execution, and output evaluation. Test attributes and execution results were also automatically registered on an Excel sheet for transparency. On average, the duration of each test case execution in Tosca was 46.5 seconds. Complete testing and reporting of the whole test scenario with 54 test cases hence required only 42 minutes. To evaluate our artefact, we referred to the evaluation framework by (Venable et al., 2016) and chose to adopt the "Human Risk Effectiveness" evaluation strategy. This evaluation strategy starts with artificial evaluations in the early design stages and quickly moves on to more naturalistic evaluations. It is suitable for user-oriented design risks and for cases where the artefact utility should be sustainable in the long run and in real situations, which were exactly the requirements of this study. In terms of the artificial evaluations, we carried out a return on investment calculation for Tosca test automation to validate its financial value for the organization in addition to the operational and strategical impacts. On one hand, we estimated the cost of ownership of automated testing based on cost factors including software licence, training, maintenance, and operations. On the other hand, we estimated the financial benefits of testing as the sum of cost savings through reduced manual testing and occurrence of production failures. While costs of manual testing can be calculated based on time requirements and hourly wages of the respective activities and staff, costs of failure recovery are more SUT-specific and have to be estimated based on the financial impact of production failure (x), the probability of prodution failure, and the degree of test automation (y). Depending on the SUT, degree of test automation y can be defined as number of automated test scenarios or utilization rate of the test automation tool. Overall, the return on investment is defined as

$$ROI = \frac{financial\,benefit - cost\,of\,ownership}{cost\,of\,ownership}$$

which is positively correlated with both x and y. In the case of this study, a positive ROI for the SUT could be already achieved with three automated test scenarios (y=3) even without recovery cost savings (x=0) due to a significant reduction in labour costs. After the artificial evaluation and validation of the artefact's ROI, we proposed the following questions (Q) for naturalistic evaluations to assess the artefact's effectiveness against the organization's problems and the following metrics (M) per question were used:

- Q: To what extent would the artefact contribute to the completion of the current test suite? M: Test coverage, confidence, and execution frequency
- Q: To what extent would the artefact improve the test process and effort? M: Test scope, effort, and effectiveness
- Q: How well would the artefact fit to the organization and support future development of the SUT? M: Perceived usability, realizability, effectiveness, and applicability to other areas of the SUT

An anonymous survey was designed with questions targeting these metrics and the responses showed an overall positive evaluation. All respondents believed that Tosca test automation could improve the efficiency of testing.

Test Debt Payback Regarding the test debt of the SUT the following test debt pay back approach was designed with the aim of structuring and organizing testing related tasks alongside feature development and operational tasks. The approach was approved by the development and management staff of the SUT. In line with the quality assurance models, the approach consists of the following components:

1. Risk-oriented prioritization of test requirements: Instead of finding test targets on the fly, test scenarios are provided by domain experts and

sorted by business risks. These high-level test scenarios are identified through the business insights of the domain experts and represent key customer-oriented functionalities of the SUT. To translate such test scenarios into concrete unit testing tasks, developers follow a process of investigation and implementation. In the investigation phase, technical components involved in the test scenario are identified, and the testability of the legacy code is assessed. If the existing components are directly testable, unit testing tasks are created in the project backlog. If refactoring of legacy code is required, testing will be carried out as a part of the refactoring tasks. Then, in the implementation phase, the created tasks are taken into sprint backlogs and assigned to developers.

2. Divide-and-conquer large testing effort: To balance the large testing related workload with the high pressure for feature development, a divide-and-conquer strategy is adopted. Since it is not feasible to tackle all the test scenarios at once, these are separated, depending on their complexity, into groups of two or three. Considering the different nature of the investigational and implementational tasks, each development cycle will only focus on one kind of task, i.e., either investigation or implementation. After one group of test scenarios is investigated in the investigation phase, the tasks are boiled down in one or multiple development cycles in the implementation phase. This approach aims to provide a standard structure for the test debt payback process, create a sense of direction for the team, and improve the efficiency by reducing coordination efforts.

3. Monitoring, evaluation, and process control: Monitoring tools are adopted for increasing transparency of the progress. Code quality and test coverage related metrics are continuously measured and integrated into the standard reports. Tests created by subcontracted developers are reviewed and accepted by an internal technical lead, and conformance to coding standards and the established testing process is controlled by the project manager. Testing activities are integrated into all existing organizational activities, especially the sprint planning, review, and retrospective.

The test debt payback approach was presented to and accepted by both the development team and the business stakeholders in the SUT's organization. An anonymous survey was conducted with the same group of stakeholders to evaluate the approach. While all respondents agreed on the importance and necessity of unit testing for the SUT and were generally confident about the applicability of the approach, they had different expectations regarding the amount of workload during the described process.

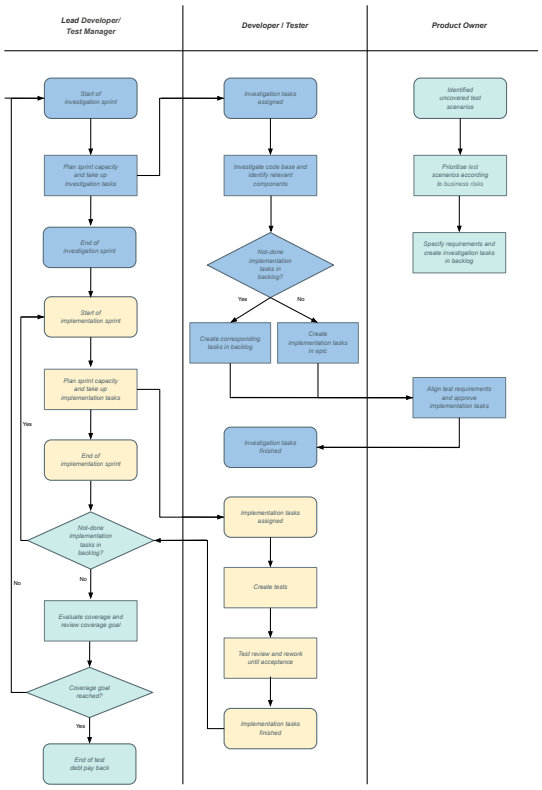

Figure 2. Test debt payback process

5. Discussion

Through ten weeks of close collaboration with the SUT's development and management team, this study evaluated and designed a customized quality assurance strategy for the SUT's organization. Relieving human testers and removing development bottlenecks, test automation not only increased product confidence but also contributed to process improvements. However, implementing test automation was not a simple task. The SUT's organization faced serious challenges coming from various organizational levels. In response to these challenges, we designed and delivered concrete technical implementations as well as strategic plans and approaches, including some outside the scope of this paper. On one hand, automated E2E

tests were implemented for the SUT with Tricentis Tosca. Hereby we aimed to validate the financial and operational benefits of test automation which are prevalent in the literature. First, from a financial standpoint, our internal ROI calculation illustrated the financial impact of test automation tool adoption in relation to its degree of utilization and the business risks of the SUT. While there exist test automation ROI calculation tools[2], our calculation also took the potential savings through prevented product failures into account. Hence it quantified the rationale that more business-critical applications would benefit more from test automation. This is in correspondence with the checklist developed by (Garousi and Mäntylä, 2016), where "SUT is mission critical" is a factor favouring test automation. While test automation may require ongoing maintenance and substantial initial effort for implementation and integration, there is no doubt that it yields a positive return on investment (ROI) considering its criticality in our case. However, it is recommendable that organizations regularly review the ROI calculation to improve its accuracy and pay attention to the actual empirical performance of the tool. Second, the test automation tool Tricentis Tosca provides powerful features in accordance with the state-of-the-art test automation practices. For instance, the script-less modular framework and data-driven testing functionalities enable quick and simple test creation as well as easier maintenance of existing tests, which significantly decrease the estimated effort for E2E testing. In addition, when developers have limited capacity for testing, codeless E2E tests could be a great help since they do not consume development time. Finally, we could also confirm the positive operational impact of test automation through the qualitative evaluations collected from the stakeholders. However, one discrepancy existed regarding the estimated effort after adoption of the test automation tool. This discrepancy was also observed from the qualitative evaluation of the test debt payback approach. While the operational staff as well as upper management of the organization were convinced of the benefits of test automation and organized approaches, the short duration of this study did not suffice to observe actual implementations nor showcase the long-term impact of the designed artefacts. Establishment of quality assurance inevitably disrupts accustomed workflows and brings additional effort; hence, teams must plan operational tasks and clarify their ownership at the very start. The major concern of employees about time and resource allocation, prioritization, and upper management approval again stresses the importance of leadership commitment as well as its transparent vertical communication. While upper management requires information and motivation from the operational staff to grant resources, operational staff requires affirmation from the upper management to prioritize specific tasks. We hence promote the necessity of transparency regarding decisions such as budget, roadmaps, and upper management opinions. Our experience so far validates and complements existing research (Garousi and Mäntylä, 2016; Wiklund et al., 2017).

Despite the overall positive results of the evaluations, the design process faced various unexpected difficulties that reflected the general challenges of implementing software quality assurance in real-life scenarios. First, the design process was confronted with constant conflicts between the abstractness of the quality assurance models and the specificity of organizational characteristics and constraints. Despite the large amount of available literature, tools and guidelines provided by the research and SQA communities, the actual implementation of such theoretical knowledge in organizations usually consists of strategic and operational decision-makings that require very specific tailoring to the team's technical, organizational, and environmental conditions. General models and dictates of best practices are not particularly helpful when certain limitations are hard to conform to. Especially in the short term, compromises are almost impossible to avoid. Either dedicated resources are allocated for SQA or the team must acquire a high level of consciousness and discipline to remove such constraints. It is possible that for many teams, like the one in this study, the allocation of resources has its limits. In such cases, motivated employees who consistently initiate discussions and push forward positive changes as well as upper-management support are crucial for long term success. Second, the organization had to make trade-offs between different goals and best practices. For feature development, practices from the agile development model Scrum were adopted by the SUT team. However, limited budget precluded the Scrum master role. The small number of developers in face of the highly complicated SUT and large number of operational tasks further dragged the team towards a less agile direction. A propensity of cost savings had already caused the project to choose the most economical subcontractor for the initial development of the SUT, which then backfired with high amounts of technical debt and large correction efforts. This validated the existing research that investment in software quality should not occur towards the end of the project (Slaughter et al., 1998). With many hierarchy layers between the project manager and the

[2]http://www.elbrus.com/services/test_automation_roi_calc/

resource allocator, communication effort was high and upper-management engagement was difficult to achieve. With a matrix project organization structure, internal employees including the project lead came from different departments and participated part-time on the project with full-time subcontracted developers. Subcontracting development and cross-continental separation of stakeholders were also factors that led to obscurities regarding dependencies and requirements and increased coordination efforts. As remote collaboration and subcontracting become more and more prevalent in the software industry, we argue that such factors could also play a role in the establishment of SQA.

6. Limitations and Future Research

This study contributes to the field of software quality assurance by validating and strengthening existing research through empirical evidence obtained from a non-trivial real-world scenario. It provides insights from the evaluation and design of test automation in a corporate environment, especially regarding the organizational mechanisms illustrated by the conceptual framework of causes and solutions. The framework and its discussions, despite stemming from the SUT's organization, offer enough abstraction for generalization. In addition, our approach to test debt pay back is also of referential value for industry practitioners, in particular for resource-scarce development teams. Nonetheless, this study has several limitations that future research should tend to. First, the short time frame limits the scope of evaluation and the conduct of further design cycles with the SUT. To properly examine the effect of the artefacts on the quality of the SUT, more artificial evaluations are necessary after actual implementations. For example, test coverage metrics, number of reported incidents, static code quality metrics are all performance indicators that should be observed for the test strategy to be continuously improved. Second, our own technical and business insights were limited. By obtaining deeper knowledge of the SUT's technical construct and through direct participation in unit or integration testing, the scope of design and evaluation could be further improved. Better knowledge of trading mechanisms and customer relations could also help us gain deeper insights about the organizational impact of SQA. Further research should hence conduct long-term studies and work in closer collaboration with the development teams. Finally, the sample size of one SUT in one organization could limit the transferability of our conclusions. Other software products could have lower levels of risks than the SUT, and other organizations could possess very different cultures. The B2B online trading platform may not be representative of other complex software systems. All these properties would affect an organization's attitude and approach towards SQA and test automation. In this regard, studies that compare software products within the same organization, or similar products from different organizations could be more insightful. Furthermore, research that validates our artefacts, such as the conceptual framework or test debt payback approach, by applying them on other projects and analysing the cost-benefit trade-offs of the proposed framework, would be greatly appreciated.

References

Ateşoğulları, D., & Mishra, A. (2020). Automation testing tools: A comparative view. *International Journal on Information Technologies & Security*, *12*(4), 63–76.

Engström, E., Storey, M.-A., Runeson, P., Höst, M., & Baldassarre, M. T. (2020). How software engineering research aligns with design science: A review. *Empirical Software Engineering*, *25*(4), 2630–2660.

Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., & Juristo, N. (2017). A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, *43*(7), 597–614.

Garousi, V., & Mäntylä, M. V. (2016). When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology*, *76*, 92–117.

Gregor, S., & Hevner, A. R. (2013). Positioning and presenting design science research for maximum impact. *MIS quarterly*, 337–355.

Helio Yang, Y. (2001). Software quality management and ISO 9000 implementation. *Industrial Management & Data Systems*, *101*(7), 329–338.

Hevner, A., R, A., March, S., T, S., Park, Park, J., Ram, & Sudha. (2004). Design science in information systems research. *Management Information Systems Quarterly*, *28*, 75–.

Humphrey, W. (1988). Characterizing the software process: A maturity framework. *IEEE Software*, *5*(2), 73–79.

IEEE. (2014). IEEE standard for software quality assurance processes. *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, 1–138.

ISO/IEC/IEEE. (2018). *Software engineering — guidelines for the application of ISO 9001:2015 to computer software* (ISO 90003:2018(E)). International Organization for Standardization.

Jamil, M. A., Arif, M., Abubakar, N. S. A., & Ahmad, A. (2016). Software testing techniques: A literature review. *6th international conference on information and communication technology for the Muslim world (ICT4M)*, 177–182.

Laukkanen, P. (2006). Data-driven and keyword-driven test automation frameworks. *Master's thesis. Helsinki University of Technology*.

Ng, S. P., Murnane, T., Reed, K., Grant, D., & Chen, T. Y. (2004). A preliminary survey on software testing practices in australia. *Australian Software Engineering Conference. Proceedings.*, 116–125.

Parzinger, M. J., & Nath, R. (2000). A study of the relationships between total quality management implementation factors and software quality. *Total Quality Management*, *11*(3), 353–371.

Paulk, M., Curtis, B., Chrissis, M., & Weber, C. (1993). Capability maturity model, version 1.1. *Software, IEEE*, *10*, 18–27.

Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., Hui, W., Virtanen, V., & Bragge, J. (2020). Design science research process: A model for producing and presenting information systems research. *arXiv preprint arXiv:2006.02763*.

Slaughter, S. A., Harter, D. E., & Krishnan, M. S. (1998). Evaluating the cost of software quality. *Communications of the ACM*, *41*(8), 67–73.

Solis, C., & Wang, X. (2011). A study of the characteristics of behaviour driven development. *37^{th} EUROMICRO conference on software engineering and advanced applications*, 383–387.

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, *1*.

Torkar, R., & Mankefors, S. (2003). A survey on testing and reuse. *Proceedings 2003 Symposium on Security and Privacy*, 164–173.

Umar, M. A., & Chen, Z. (2019). A study of automated software testing: Automation tools and frameworks. *International Journal of Computer Science Engineering (IJCSE)*, *6*, 217–225.

Venable, J., Pries-Heje, J., & Baskerville, R. (2016). Feds: A framework for evaluation in design science research. *25*(1), 77–89.

Walkinshaw, N. (2017). *Software quality assurance: Consistency in the face of complexity and change*. Springer.

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer Berlin Heidelberg; Imprint; Springer.

Wiklund, K., Eldh, S., Sundmark, D., & Lundqvist, K. (2017). Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability*, *27*(8), e1639.

Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P., & Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. *OSDI*, *10*, 2685048–2685068.