Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Technologies

Bachelor Thesis

Automated regression testing of web application

Kelgenbayev Bekzat

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Bekzat Kelgenbayev

Informatics

Thesis title

Automated regression testing of web application

Objectives of thesis

The primary objective of this thesis is to evaluate and compare the efficiency of automated and manual testing, utilizing predefined criteria to establish which approach offers superior performance in particular test scenarios.
Partial objectives:
- Conduct literature review using available information sources focusing on the topics of web application testing and test automation.
- Analyze chosen web application, specify its functional requirements, and develop a set of test cases.
- Execute the test cases using both manual and automated testing methods to ensure proper functionality and identify any defects or errors.
- To compare the results of the manual and automated testing methods based on the selected criteria and provide recommendations for future testing.

Methodology

The theoretical part of this thesis consists of an analysis of professional literature and online sources in the area of test automation. This section will outline various test groups, levels, and types of testing, along with the different tools used for automated testing. The insights gained from the theoretical part will be applied in the practical part, which will involve analyzing a chosen web application, specifying requirements for testing, and developing a set of test cases. Both manual and automated testing will be implemented, evaluated, and compared. The results of the practical part will be used to formulate recommendations and discuss the advantages of the proposed automation. Thesis conclusions will utilize knowledge obtained from both theoretical and practical parts.

The proposed extent of the thesis

50-60

Keywords

Test automation, manual testing, regression, web-application, test case

---

Recommended information sources

Dorothy Graham, Mark Fewster. 2019. ”Experiences of Test Automation: Case Studies of Software Test Automation”

Jonathan Rasmusson. 2019. ”Test Automation in the Real World: Practical Lessons for Automated Testing”

Jürgen Münch, Frank Padberg. 2017. ”The Impact of Test Automation on Software Quality”.

Sriram Papineni. 2020. ”Practical Test Automation: Build automated test cases from scratch”

Veronika Pramukova, Pavol Návrat. 2020. ”The Impact of Test Automation on Software Development Efficiency”

Expected date of thesis defence

2023/24 SS – PEF

The Bachelor Thesis Supervisor

Ing. Jan Pavlík, Ph.D.

Supervising department

Department of Information Technologies

Electronic approval: 4. 7. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Head of department

Electronic approval: 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 11. 03. 2024

Declaration

I declare that I have worked on my bachelor thesis titled " Automated regression testing of web application" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on 15 March 2024 _____

Acknowledgement

I would like to thank my supervisor Ing. Jan Pavlík, Ph.D. for his support during my work on the Bachelor Thesis.

Automated regression testing of web application

Abstract

This study investigates the integration of automated testing in software development, comprising both theoretical insights and practical implementation. The theoretical part provides a detailed literature review on software testing, encloses functional and non-functional testing types and classifications, additionally, it describes the fundamentals of test automation tools.

The practical aspect unfolds with an introduction to an internal web application, followed by the establishment of testing requirements and the manual testing process. The study then searches for insights into the impact of automated testing on software quality. The research concludes by summarizing key findings, emphasizing the importance of the automated testing approach in modern software development practices.

Keywords: Test automation, manual testing, regression, web application, test case, test requirements, software, Gherkin.

Automatizované regresní testování webové aplikace

Abstrakt

Tato práce zkoumá integraci automatizovaného testování ve vývoji softwaru, zahrnuje jak teoretické poznatky, tak praktickou implementaci. Teoretická část poskytuje podrobný literární přehled o testování softwaru, zahrnuje funkční a nefunkční typy a klasifikace testování, dále popisuje základy nástrojů pro automatizaci testování.

Praktická část je založena na automatizovaném testování interní webové aplikace. Po stanovení požadavků na testování je proveden proces ručního a následně automatizovaného testování. Studie poté hledá poznatky o dopadu automatizovaného testování na kvalitu softwaru. V závěru jsou shrnuta klíčová zjištění a zdůrazněna důležitost přístupu automatizovaného testování v moderních postupech vývoje softwaru.

Klíčová slova: Automatizace testování, ruční testování, regrese, webová aplikace, testovací případ, požadavky na testování, software, Gherkin.

Table of content

1 Introduction

Programming has transitioned from being a niche industry to an integral part of our daily lives. It permeates everything from the apps on our smartphones to the complex systems driving businesses, finance, agriculture, logistics, and medicine, making software development a pivotal force in shaping our world. Ensuring the accurate execution of program operations depends on one key aspect: software testing. At the core of software engineering, software testing upholds the quality of the final product.

Software testing is not a field that captures significant public attention. Initially, it emerged as a foundational element of the development process. The integration of automated testing has been present since the inception of testing practices. However, only recently has the potential of automated testing truly accelerated. In spite of extensive academic exploration into testing and automation, the impact of automation advancements on testing professionals has largely remained unexplored.

This thesis aims to investigate and contrast two testing techniques, identifying their advantages and limitations, and subsequently making a comparative analysis. The research seeks to add to the existing reservoir of knowledge regarding automated software testing, providing certain insights that could help in the decision-making processes.

The primary goal is to examine and assess diverse approaches testing to complex systems while enhancing software testing quality, reducing defects, and improving the efficiency of web application development.

2 Objectives and Methodology

2.1 Objectives

The primary objective of this thesis is to evaluate and compare the efficiency of automated and manual testing, utilizing predefined criteria to establish which approach offers superior performance in particular test scenarios.

Partial objectives:

- Conduct a literature review using available information sources focusing on the topics of web application testing and test automation.

- Analyse the chosen web application, specify its functional requirements, and develop a set of test cases.

- Execute the test cases using both manual and automated testing methods to ensure proper functionality and identify any defects or errors.

- To compare the results of the manual and automated testing methods based on the selected criteria and provide recommendations for future testing.

2.2 Methodology

The theoretical part of this thesis consists of an analysis of professional literature and online sources in the area of test automation. This section will outline various test groups, levels, and types of testing, along with the different tools used for automated testing. The insights gained from the theoretical part will be applied in the practical part, which will involve analysing a chosen web application, specifying requirements for testing, and developing a set of test cases. Both manual and automated testing will be implemented, evaluated, and compared. The results of the practical part will be used to formulate recommendations and discuss the advantages of the proposed automation. The Thesis conclusions will utilize knowledge obtained from both theoretical and practical parts.

3 Literature Review

3.1 Software Testing Overview

Software testing is like quality control for software. Just as you wouldn't want a faulty product to reach customers, you don't want faulty software to be used by users. Testing is the process of examining a software application to find defects and ensure that it works as intended. It is used to verify that functional requirements were met. The major purpose of verification and validation activities is to ensure that software design, code, and documentation meet all the requirements imposed on them (Lewis, 2008, p. 10). It's an essential element of creating reliable, user-friendly, and high-quality programs.

When a company contemplates a new development project, it prepares a business case that clearly identifies the expected benefits, costs, and risks (Everett, McLeod, 2007, p. 9). Upon input, the tester is provided with a program slated for testing, along with corresponding requirements. While observing the program under certain conditions, the tester obtains information about whether the program is compliant or non-compliant with the requirements through the output. This gathered information serves as the basis for fixing any detected bugs within an already established product or for adjusting the requirements of a product that remains in the developmental phase.

Software testing involves the validation of congruity between the real-world operational performance of a program and its expected behaviour. This process is conducted across a limited collection of tests, meticulously chosen through a specific methodology.

Tests can vary in duration, ranging from short assessments to more extended evaluations like performance tests that check a system's response under sustained loads.

Consequently, within the testing phase, the tester undertakes two primary responsibilities:

• Supervising the execution of the program and generating simulated scenarios to test its behaviour.

• Examining how the program behaves in different scenarios it generates and then contrasting its observed behaviour with anticipated outcomes.

Testing identifies defects and failures and provides information on the software and the risks associated with their release to the market (Homès, 2012, p. 5). But, if we consider the tasks of modern testing, we can conclude that they consist not only of detecting errors in programs but also of identifying the reasons why they occur. This approach to the testing process allows developers to perform their work with maximum efficiency, eliminating the detected errors quickly and promptly.

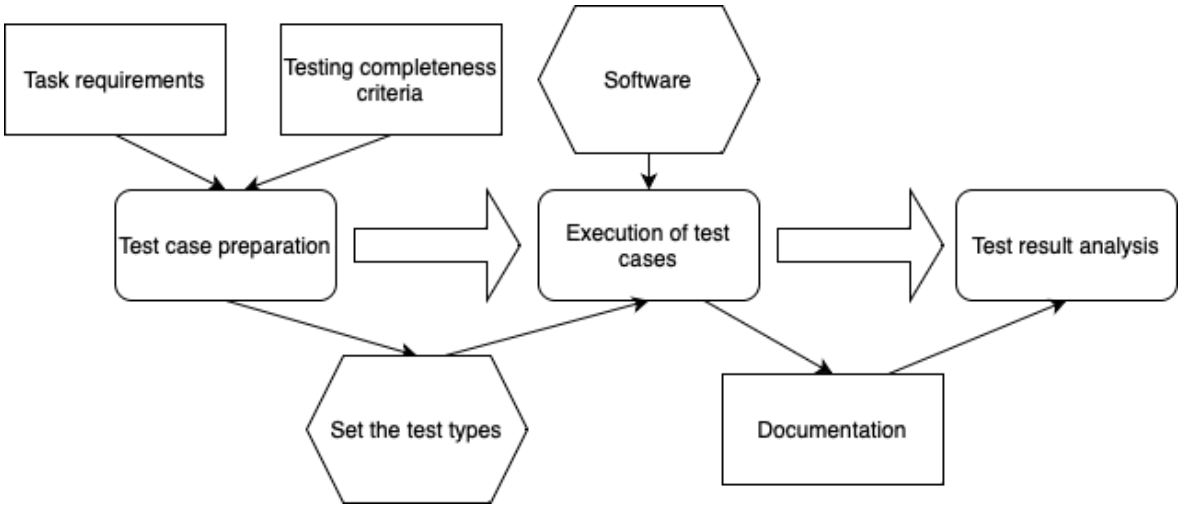

Figure 1: High-level Overview of Software Testing, Source: Homès, 2012

3.2 Classification of Types of Testing

Tests differ relevantly in the tasks they solve, and the technique used. The testing methods are used to comprehensively evaluate various aspects of a software application, covering from its features and user interface to its performance, security, and additional areas. In this chapter we will discuss the different types of testing that serve distinct purposes, addressing a wide spectrum of requirements and concerns.

3.2.1 Functional Testing

Functional testing is achieved by a series of tests that exercise increasingly more of the software that directly enables users to accomplish this routine daily business (Everett, McLeod, 2007, p. 99). This type of testing is performed to validate that the software is working as expected and meets the user's requirements. The primary focus is on the software's functionality and includes various levels of testing such as unit testing, integration

testing, system testing, and acceptance testing (Leloudas, 2023, p. 5). These levels are widely known as the testing pyramid (shown in Figure 2), a concept introduced by Mike Cohn.

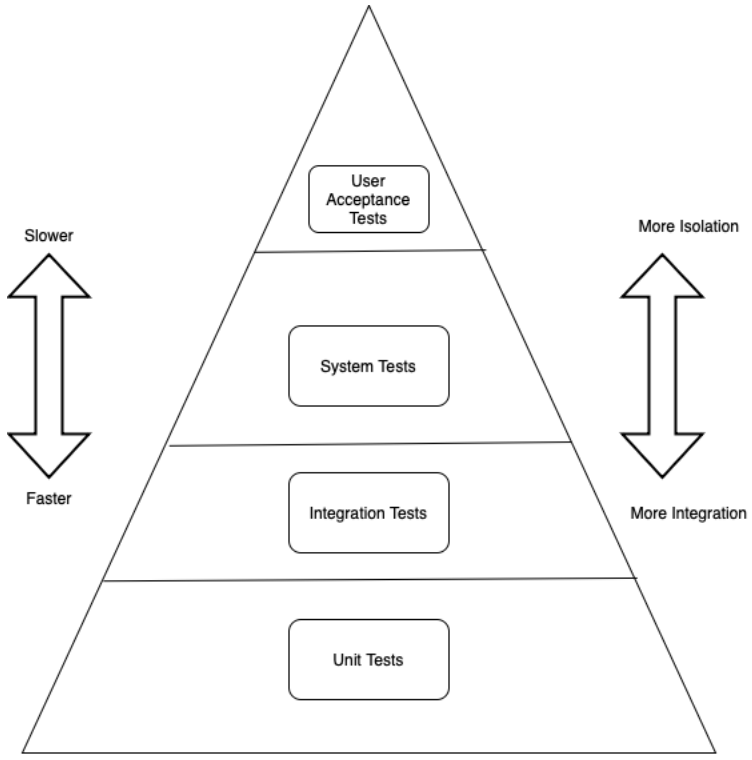

Figure 2: Testing Pyramid, Source: Leloudas, 2023

3.2.2 Non-Functional Testing

Non-functional testing customizes settings programmatically to align with custom requirements. The objective is to ensure that the software meets its non-functional requirements and provides a positive user experience (Leloudas, 2023, p. 2). Thus, non-functional testing is the testing of all properties of the program, regardless of the functionality of the system. Such estimates can be obtained in terms of parameter estimation as:

1. Reliability (ability of the system to respond to unforeseen situations).
2. Performance (the ability of the system to work under heavy loads).
3. Convenience (study of user convenience with the application).
4. Scalability (the ability to scale the application both vertically and horizontally).
5. Security (detection of potential application breaches and high availability theft).
6. Portability (the ability to use on the set of platforms used)

3.2.3 User Interface Testing

This is testing the correctness of the representation of the elements of the user's view on various solutions, the correctness of their achievement for the users to perform various tasks, and a reasonable assessment, a universal setting of the task for themselves. This testing makes it possible to evaluate how effectively the user can work with the application and how the appearance is created using actual documents by designers. When checking the user interface, the main task of the tester is to identify visual and structural flaws in GUI applications, applicability, and ease of navigation in the application, and adjust the application to process data using other input devices with keyboards, mice, and input devices. User interface testing is necessary to make sure that the interface meets the requirements and extension and enhances the user experience with the graphical interface of applications.

3.2.4 Usability Testing

Usability testing is a type of testing performed to evaluate how user-friendly a product is. The objective is to determine how easy it is for users to learn and use the product, and how efficient they are at completing tasks (Leloudas, 2023, p. 20). Usability testing is a method of evaluation, the speed of assessing the degree of usability of applications, training users when working with open, as well as reasonable users of the product being developed, which corresponds to its understandability and relevance in given conditions. Such testing is necessary to get a great positive user experience when working with the application.

3.2.5 Security Testing

Security testing determines the security of the software product, identifying Major Vulnerabilities Computer systems often encounter cyberattacks that reveal violations of system detection or theft of confidential data (Homès, 2012). Security testing provides an opportunity to evaluate the actual verification and effectiveness of the mechanisms used in the system in penetration investigation. In the process of security testing, the tester performs the same actions that a real cracker performs. Hacking the security system with a special tool; logins and passwords are detected using external sources; generation of errors to identify cases of detection in the system in the process of its recovery; use of disclosed unpatched system vulnerabilities.

3.2.6 Installation Testing

Install ability testing involves evaluating the ease of installing and configuring software, ensuring that it complies with installation requirements and specifications. This term means testing the correctness of the installation (installation) of a certain software product (Leloudas, 2023, p. 49). Such testing usually takes place in artificially created environments in order to determine the likelihood of a threat to the operation. The main reasons for conducting such tests involve checking the correct behaviour of the product during deployment or upgrade. Frequency and stable installation require very important control over product consumption, as users allow users to consume resources faster and more efficiently while maintaining the correct behaviour of this product in all tested software environments.

3.2.7 Configuration Testing

Configuration testing is designed to evaluate software performance under various system configurations and involves testing the system under different configurations to determine how performance is affected by different settings (Leloudas, 2023, p. 15). Depending on the type of software product being tested, configuration testing can have different goals. Usually, this is either determining the optimal hardware configuration that provides sufficient performance parameters for the software to work or checking a specific hardware configuration (or a platform that includes, in addition to hardware, third-party software required for the program to work) for compatibility with the product being tested. When it comes to client-server software, configuration testing is carried out separately for the server and separately for the client. Usually, when testing the compatibility of a server with a certain configuration, the task is to find the optimal configuration, since the stability and performance of the server are important. When testing a client, on the contrary, they try to identify software flaws in any configurations and, if possible, eliminate them.

3.2.8 Stress Testing

Stress Testing refers to determining the application's stability under extreme loads and allows you to identify the maximum number of tasks of the same type that the program can perform in parallel. It pushes the application beyond its normal operating conditions to evaluate how it responds to these conditions (Leloudas, 2023, p. 14). The most popular goal of load testing in the context of client-server applications is to estimate the maximum number

of users who can simultaneously use the services of the application. Testing reliability and recovery after failures (stress testing). This type of testing is often done for software that works with valuable user data, the continuity of operation, and the speed of recovery from failures which are critical for the user. Failure and recovery testing tests the program's ability to recover from hardware failure quickly and successfully, network outages, or critical errors in the software itself. This makes it possible to assess the possible consequences of a failure and the time required for the subsequent restoration of the system. Based on the data obtained during testing, the reliability of the system can be assessed, and, in case of unsatisfactory performance, appropriate measures can be taken to improve recovery systems.

3.2.9 Localization Testing

Localization testing makes it possible to find out how well the product is adapted to the population of certain countries and how it corresponds to its cultural characteristics. Usually, cultural, and linguistic nuances are considered, namely the translation of the user interface, related documentation, and files into a certain language, and the correctness of the formats of currencies, numbers, times, and phone numbers is also tested.

3.2.10 Stability Testing

Stability testing checks the performance of the application during long-term use at medium loads. Depending on the type of application, certain requirements are formed for the duration of its uninterrupted operation. Stability testing seeks to identify application issues such as memory leaks, severe load spikes, and other factors that can prevent the application from working for an extended period.

3.2.11 Volume Testing

Volume testing is used to evaluate how well the application can handle large volumes of data and it aims to gauge the application's performance across different data volumes (Leloudas, 2023, p. 15). The task of volume testing is to identify the reaction of the application and evaluate possible deterioration in the operation of the software with a significant increase in the amount of data in the application database. This testing includes:

1. Measurement of the execution time of operations related to obtaining or changing database data at a certain intensity of requests.

2. Identification of the dependence of the increase in the time of operations on the amount of data in the database.
3. Determination of the maximum number of users who can simultaneously work with the application without noticeable delays from the database.

3.2.12 Scalability Testing

Scalability testing is a type of software testing designed to test the ability of a product to increase (sometimes decrease) the scope of certain non-functional features (Leloudas, 2023, p. 15). Some types of applications must be easily scalable and, of course, remain operable and withstand a certain user load.

3.3 Regression Testing: Concepts and Importance

Regression testing is a pivotal process within software development that involves observing a product for new bugs or flaws after implementing changes. This procedure holds great significance in maintaining product integrity. Running regression tests is often the most time-consuming testing activity. When software is changed or extended, checking that the existing functionality still works correctly is at least as important as checking that the new functionality works correctly (Bierig, Brown, Galván, Timoney, 2021, p. 297). The following points elucidate the essence and significance of regression testing:

1. Degradation Prevention: When making changes to code, there is a risk that the new code can be used on existing functionality. Regression testing allows you to detect such negative impacts and prevent product degradation.
2. Ensuring Stability: The regression testing process helps ensure that previously fixed bugs are not returned, and the product remains stable and reliable.
3. Maintaining Functionality: Regression testing assumes that changes made do not break Existing Functionality. This is especially important in evaluating targeted product support.
4. Save time and resources: Automating regression testing allows you to exploit your application quickly and efficiently for defects. This saves time and resources for busy and testers.
5. User Trust: Regression testing of glucose for a reliable and quality product. This indicates user trust and satisfaction with the application.

In general, a regression measurement involves tracking the quality markers of a product across its entire lifespan. Its purpose is to guarantee the dependability, consistency, and assurance of applications following each modification. The testing range is divided into our examination subjects and a methodology for sequencing tests according to a risk management system. The priorities are set from 1 (highest) to 4 (lowest). Priorities 1 and 2 are the main candidates for test automation. These priorities are subject to change based on feedback from support, customers, and others. As a basis for our test objects, we use the use cases and requirements and link them to test objects and test scripts so that we can build up a complete traceability chain (Graham, Fewster, 2012).

3.4 Test Automation

Automated testing is validating a software solution using a specialised software tool, and this typically involves automating functions as part of the testing process (Jose, 2021, p. 8). Automating tests results in a significant reduction in the effort and energy required for thorough system testing. It guarantees extensive and uniform test coverage during execution, ensuring comprehensive validation of major data sets to suggest stakeholder confidence. Employing test automation can progressively decrease the dependency on manual testing and manual testers over time.

You can evaluate the program at different levels of testing: code (including unit tests and integration testing), API, and GUI. Different types of tests are more suitable for different scenarios. Let's delve into the various types of automated testing.

Automated testing can be categorized as follows:

- Unit test—A test that examines the behaviour of a distinct unit of work. A unit of work is a task that is not directly dependent on the completion of any other task (Tudose, 2020, p. 5). Unit Testing is a script that tests a specific code by means of initializing it, calling the different methods and functions in the code, and checking the values returned by the method or function (Kaul, 2022, p. 22). Developers typically write unit tests locally as part of standard Test-Driven Development (TDD) practices. While most unit tests are integrated into the software code itself to evaluate the application's code, automation engineers or testers may also be involved in their creation.

- API Testing. Any software system that is based on API architecture can implement API testing. This type of testing is meant to validate the business layer of the software system by inspecting the response time of the APIs used in the application for all the requests (Kaul, 2022, p. 22).
- UI Testing. This phase is the last phase of testing in the case of a large group of software projects. The UI is what the customers/users interact with and hence is an important part of the testing process (Kaul, 2022, p. 22).

A unit test usually supplies varied inputs to a function and validates its expected output. For example, a phone number validation function would be tested with predetermined numbers to confirm its accuracy. Similarly, a quadratic equation solver function could be tested by pre-calculating equations and comparing the returned roots.

Unit tests excel at assessing logic-heavy code. In cases where code relies heavily on calls to other classes and lacks substantial logic, unit tests might be challenging to develop. APIs represent callable functions for data retrieval. Testing APIs involves sending prepared requests and comparing the responses against expectations. This approach is applicable whenever an application possesses an API.

GUI testing pertains to the graphical interface presented to users. This form of testing is the most intricate. For instance, when assessing website functionality, emulation of browser behaviour is required. This process involves analysing displayed information. GUI testing, also known as End-to-End (E2E) or acceptance testing, is essential since it mimics user interactions.

In summary, the following scenarios warrant automation:

1. Testing hard-to-reach system components (backend processes, logging, database operations).
2. Frequently used functionality with high error risks. Automated testing of critical functions ensures prompt error detection and resolution.
3. Automating routine tasks, such as data sorting and form field validation.
4. Validation message testing (filling fields with incorrect data to check for appropriate validation messages).
5. Long End-to-End scenarios.

6. Precise mathematical calculation validations.
7. Verification of accurate data search functionality.

Why Test Automation?

- Advanced test coverage and enhanced testing quality – Utilizing test automation can expand the scope of testing by conducting assessments on a significant volume of data. This automation also ensures uniformity and precision in test execution by eliminating the potential for human errors.
- Enhanced efficiency and efficacy – Test automation allows for the repetition of identical tests or variations of tests with different inputs, requiring less expenditure of resources and effort. This leads to increased overall testing efficiency.
- Improved dependability and precision – Test automation outperforms manual testing in terms of reliability, as it minimizes the likelihood of human errors during test execution.
- Economical and time-saving – As an example, automated regression testing facilitates the swift execution of regression tests, enabling them to be conducted frequently, across various environments, without human supervision, and with extensive datasets. This ultimately reduces costs and conserves testing resources.
- Test automation enriches the testing process by enabling unattended execution, and decreasing reliance on specific timing, location, and resources.

3.5 Test Automation Tools and Frameworks

Test automation tools and frameworks are essential components of the software testing process. They help streamline and enhance testing activities by automating repetitive tasks, increasing efficiency, and providing better coverage of test scenarios. Here’s an overview of test automation tools and frameworks:

3.5.1 Selenium

Selenium, an open-source tool, serves as an interface for automating user interactions on web browsers. Selenium's setup offers great flexibility, contributing to its widespread popularity among test and software engineers. The tool is highly favoured due to the built-in support for Selenium testing in most contemporary browsers, adding to its convenience. Similar to many other open-source tools, Selenium relies on community contributions for

support, facilitated through third-party libraries. Over the past decade since its open-sourcing, users have consistently made improvements, enhancing its capabilities (Sambamurthy, 2023).

Selenium comprises three key components for test creation and orchestration:

- WebDriver, a vital element responsible for test execution by invoking browser-specific drivers. WebDriver provides an API with language bindings for Python, Ruby, Java, and C# and supports integrations like Cucumber and TestNG.
- Grid, a server optimizing test runtime by distributing commands across multiple remote browser instances. Grid's hub and node manage requests from WebDriver, executing them on remote devices. It dynamically scales for varying test runs.
- The third component is IDE, a record-and-playback tool available as a browser plugin for Chrome and Firefox. While useful for quick checks, the generated code is typically non-reusable, making it less suitable for larger projects.

3.5.2 Jenkins

Jenkins is an open-source automation server widely used for continuous integration. It facilitates the automated building, testing, and deployment of software projects, providing a framework for developers to integrate code changes more efficiently and consistently.

Main advantages of using Jenkins:

- Continuous Integration. Jenkins is primarily known for its CI capabilities. It automates the process of building and testing code changes whenever developers commit code to version control repositories.
- Extensibility and Plugin Support: Jenkins has an extensive plugin ecosystem, allowing users to customize and extend its functionality to suit specific needs. Plugins cover a wide range of areas, from source code management and build tools to deployment and monitoring.
- Distributed Build Support: Jenkins can distribute builds across multiple machines, enabling parallel execution of jobs.

3.5.3 Tosca

Without properly identifying web application elements, the test automation code will not be able to run properly, since it can't find the elements on the web pages and perform actions on them (Kinsbruner, 2022, p. 125). Tosca is a software testing tool developed by Tricentis, but it is primarily known for its capabilities in end-to-end functional testing and test automation. While it does offer features like dashboards, analysis, and integrations, the claim that it demands only basic technical knowledge might be a bit optimistic. Tosca can indeed be used for functional testing, including web applications, and it supports mobile testing as well. It provides features for creating, managing, and executing automated tests across different platforms and technologies.

3.5.4 JUnit

JUnit is an open-source framework for writing and executing unit tests in Java (Hightower, Lesiecki, 2001). It plays a crucial role in supporting Test-Driven Development (TDD) practices, where tests are written before the actual code implementation. JUnit provides a simple and effective way for developers to ensure the correctness of their code by automating the testing process. The design part of TDD is more important than the testing part. When we do test last, we don’t reap the design benefit from the practice (Gulati, Sharma 2017, p. 2).

Widely embraced by the Java development community, JUnit has become a standard for unit testing. Its popularity is attributed to its simplicity, and ease of integration with various IDEs (Integrated Development Environments).

3.5.5 NUnit

NUnit, a robust unit testing framework bespoken for the .NET platform, stands out for its versatile features and widespread adoption among developers.

One of its notable features is support for data-driven testing, allowing developers to evaluate a test method with multiple sets of input data. This flexibility enables comprehensive testing across diverse scenarios, ensuring the code's reliability under various conditions.

Unit testing isn’t designed to achieve some corporate quality initiative; it’s not a tool for the end-users, or managers, or team leads. Unit testing is done by programmers, for programmers (Hunt, Thomas, Hargett, 2007, p. 2). The framework's strength is further amplified by its extensive user community. This dynamic community actively contributes to NUnit's growth, sharing insights, and best practices, and providing support to fellow developers. The collective knowledge ensures that developers have access to valuable resources, discussions, and solutions to common testing challenges, fostering a collaborative and supportive environment.

Test automation tools like Selenium, Jenkins, Tosca, JUnit, and NUnit streamline software testing by automating tasks, enhancing efficiency, and facilitating comprehensive test coverage. These tools cater to various testing needs, from functional and web testing to unit testing, and contribute to the overall quality assurance process in software development.

3.6 Developers and Testers

One factor that was present in the successful automation experiences was a good relationship between the testers, test automators, and developers. A poor relationship can make automation much more difficult than it needs to be even if it still gives some benefit in the end (Graham, Fewster, 2012). If you are faced with a complex problem that you do not know how to approach, try to ask the advice of developers (Alpaev, 2017, p. 24). Essentially, developers are responsible for writing the code that forms the foundation of our software applications. Their role is to translate the requirements and design specifications into functional code. Here's why Developers are important:

- Code Quality: Developers write the code that forms the basis of our software. High-quality code ensures that the application is efficient, maintainable, and less prone to bugs and errors.
- Innovation: Developers come up with creative solutions to complex problems, leading to innovative features and functionalities that enhance the user experience and keep our products competitive.
- Speed of Development: Efficient coding practices and adherence to best practices by Developers lead to faster development cycles, which in turn allows us to deliver products to market quicker.

Testers, focus on identifying and rectifying defects, ensuring that the software meets quality standards before it's released to users. Here's why Testers are important:

- Quality Assurance: Testers thoroughly examine the software to identify any issues, bugs, or inconsistencies. This process ensures that the software meets the required quality standards and performs as expected.
- User Experience: Testers evaluate the software from an end-user perspective, ensuring that it's user-friendly, intuitive, and meets user expectations.
- Risk Mitigation: By conducting comprehensive testing, Testers help mitigate the risks associated with software failures, data breaches, and other vulnerabilities that could negatively impact our clients and the company's reputation.

Testers collaborate closely with Developers to provide early feedback, which helps in addressing issues during the development process itself. This collaborative approach saves time, effort, and resources in the long run. The collaboration between Developers and Testers is vital because:

- Early Detection of Issues: When Developers and Testers work together from the beginning, it's easier to catch and address issues early in the development cycle, reducing the cost and effort required for fixing them later.
- Effective Communication: Open and frequent communication between Developers and Testers ensures that everyone understands the project requirements, goals, and constraints. This minimizes misunderstandings and facilitates smoother development and testing processes.
- Continuous Improvement: Collaboration fosters a culture of continuous improvement. Developers learn from the feedback Testers provide, leading to better coding practices, while Testers gain insights into the development process, enabling them to create more effective testing strategies.

3.7 Types of Errors

Bugs and other software quality destroyers refer to issues or factors that can negatively impact the quality, reliability, and performance of software (Forgács, Kovács, 2023, p. 1). In the world of general-purpose, large-scale, business-critical soft¬ ware, it is not the goal of a test team to find all of the defects. That is a practical impossibility (Loveland, Miller,

Prewitt, Shannon, 2013, p. 6). They can occur at any stage of the software development life cycle, from design and coding to testing and deployment. Some common types of software bugs are:

- Functional. For example, when data is not saved. The user clicks the numerous times button, but nothing happens.
- UX defect. It refers to usability. Sometimes, to make a verification, the user needs to refresh (leave and return) to application.
- Load bug. Imagine an artificially created situation where several thousand users log into one section of a web application at the same time. If the application does not load or freezes it refers to a load bug.
- Performance bug. When a web application takes up too much space in the memory, it starts working slowly.
- Requirements bug, or logical bug. Before the development of an application began, something was not considered in the requirements. For example, costumers forgot to add a pop-up notification that the application may not work correctly when the VPN is enabled. The programmer programmed it as it was in the requirements (or as he understood them). As a result, the application works as described in the requirements, but not in the way the business needs.

While this basic bug life cycle gives you an idea, remember that real situations are more complicated. Various factors, like communication breakdowns and testing challenges, can make it trickier. Check out Figure 3 for a more detailed and comprehensive view.

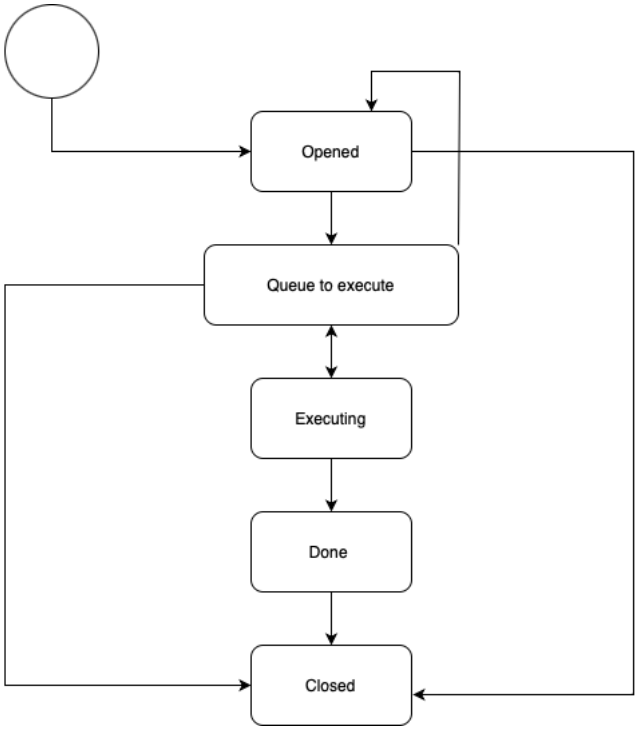

Figure 3: Basic Bug Fix Life Cycle, Source: own processing

A bug has a zero stage, when, it can be a bug, or maybe just a misunderstanding on the part of the user. When a tester encounters an inexplicable behaviour in the system's operation, they initiate a process known as localization. The primary objective of localization is to confirm whether the detected anomaly is indeed a defect. To achieve this, the tester delves into the design documentation conducts experiments, and investigates the circumstances under which the defect manifests. The tester aims to determine if the defect is reproducible and whether there are any potential workarounds.

4 Practical Part

The Practical part aims to perform manual and automated types of testing with the same business requirements for internal firm application and find out superior performance based on the following criteria:

1. Time and Efficiency
    1.1. Testing Speed
    1.2. Efficiency and Accuracy
2. Usability
    2.1. Resource Impact
    2.2. Flexibility and Adaptability

The Business requirements will be established by the clients of the firm and the testing will be carried out on an internally used within the company web application. The firm is a stock exchange organization that trades and processes financial securities and derivatives, clearing, settlement, and custody.

In order to find out the better performance of testing methods in the Software Development Life Cycle (SDLC) – the following research question was formulated and needs to be answered during practical work:

Research Question 1: *Considering the comparison between the two distinct testing approaches, which one stands out as more effective for testing a complex system?*

Research Question 2: *Is it possible for the company to completely transition from manual regression testing to automated testing for the selected web application?*

Additionally, to achieve the largest findings in this study following work steps will be performed:

- Describing the web application used for testing purposes
- Analysing the test requirements and providing a business overview
- Creation of test cases
- Manual testing
- Automated testing
- Evaluate results and compare them based on the selected criteria
- Conclusion

4.1 Introduction into Internal Web Application

Internal Web Application used by stock exchange organization called Reference Data Factory (RDF). Under the ownership of the company, RDF functions as the central repository for financial Instruments and Institutional reference data, acting as the singular source of truth for the firm and its associated entities. Its role as a business-critical application is evident in its ability to create, cancel, mature, re-activate, re-open, validate, default, and publish financial securities. The comprehensive scope of RDF encompasses a wide array of functionalities, including reporting, user management, audit data history, data source prioritization, file uploads, documentation, and workflow task management.

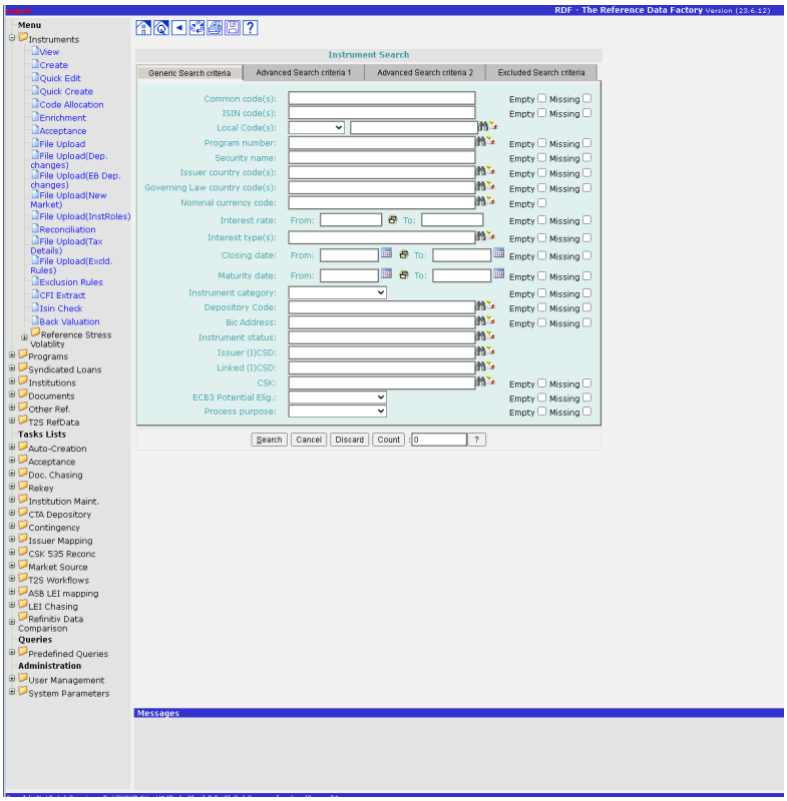

Figure 4: Reference Data Factory (RDF) GUI, Source: own processing

Web Application is often used for an accurate and timely setup of Financial Instruments and primary distribution of securities on Domestic and International markets; therefore, it is frequent adjustments are necessary to ensure that issued instruments consistently adhere to legal standards.

4.2 Establish the Requirements for Testing

The stock exchange firms are complex ecosystems where the interaction of various active members creates a dynamic and efficient financial environment. The most important component of this system is the market participants, and their role cannot be underestimated. First and foremost, market participants serve as the engine of liquidity in the market. Individual investors, institutional participants, traders, and companies create supply and demand by entering their trades, which contributes to deep and efficient liquidity. The more participants there are, the less likely large price fluctuations are, which creates stable conditions for all.

Projects and requirements for IT teams come exactly from different market participants such as banks, government, supranational organizations, investors, Allottees, and others. The role of market participants in this process is extremely important since they are the ones who have the final vision of what should be achieved and can clearly articulate their needs and expectations from the product. Customer engagement allows IT teams to better understand the context and goals of a project, which in turn leads to more effective and satisfying solutions.

To demonstrate the preparation of test cases for both manual and automated testing approaches, the following task requirements were selected.

Table 1: Client Requirements

1. Bank requests to Mature significant number of financial instruments.
2. Bank requests possibility to Re-Open the financial instruments were used.
3. Bank requests to Cancel significant number of financial instruments.
4. Bank requests possibility to Reactivate the financial instruments were used.

Source: own processing

4.2.1 High-level Overview of Business and Technical Tasks

In order to successfully create financial security in the RDF system – it must be followed next rules:

1) Received and collected a final documentation
2) Acknowledged eligibility as legal aspects
3) Allocated International Securities Identification Number (ISIN)
4) Facilitated the distribution of new securities on the RDF system

After completing these steps, the company will be ready to create a financial instrument, The created financial instrument will be assigned the status "ACTIVE", which means it is ready for active use within the RDF system and confirmation of its financial stability.

To accomplish the first and second test requirements, it will be necessary to change the status of financial instruments in the RDF system from “ACTIVE” to “MATURED”. From a business point of view, “Mature significant number of financial instruments” refers to the final liquidation date, which covers the demand to close a previously issued ISIN.

In contrast, Re-opening the financial security may happen only after the cancellation of an instrument in the RDF system. It will be only visible dynamically if the Instrument status is “MATURED”.

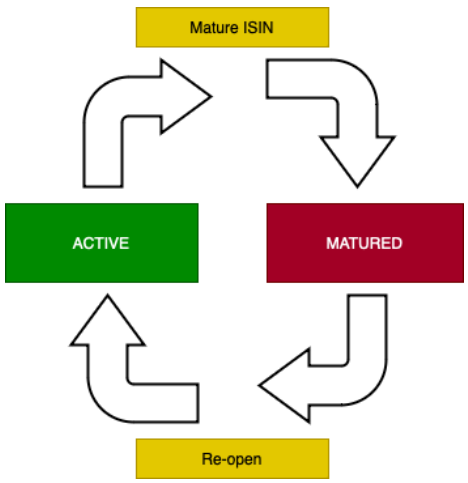

Figure 5: High-level Overview of Mature action in RDF, Source: own processing

To properly test the third and last test requirements, it will be necessary to follow the same approach. In order to “Cancel a significant number of financial instruments” it would be needed to change the status of financial instruments in the RDF system from “ACTIVE”

to “CANCELLED”. Usually, cancellation of an ISIN involves the request to void an ISIN code that was previously generated but was never introduced in the market and is no longer included in the fund documentation.

In contrast, Re-activating the financial security may happen only after the cancellation of an instrument in the RDF system. It will be only visible dynamically if the Instrument status is “MATURED”.

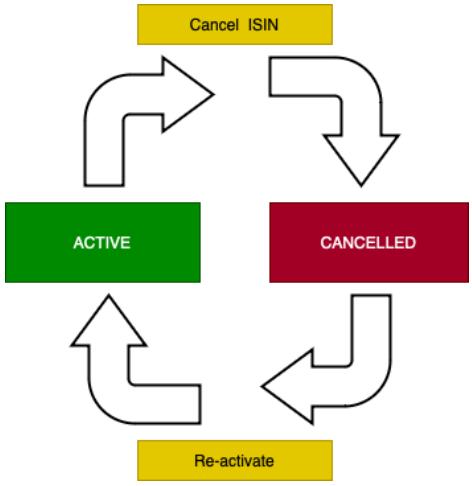

Figure 6: High-level Overview of Cancel action in RDF, Source: own processing

4.2.2 Test case Preparation

Defining test cases is an important part of testing software, where individual test situations are recognized and executed. A test case involves a set of conditions or variables that a tester uses to check whether a software system is working correctly or not. This thorough process is essential for confirming that the software aligns with the defined requirements, and it also helps identify any potential weaknesses or areas that could be enhanced in the system. Recording the test case is obligatory; document relevant details such as the test case's purpose, input data, expected outcomes, and the procedural steps to be followed during execution. This accurate documentation not only serves as a reference but also plays a key role in enhancing the overall testing process. By having a well-documented test case, it becomes easier for team members to understand and replicate the testing scenarios, fostering efficient collaboration and ensuring the reliability of the software.

The test scenarios outlined above have been developed to address the bank's requirements. These scenarios require both manual and automated testing.

Table 2: Test Cases for 1st Requirement

Test Requirement 1 – Mature significant number of financial instruments				
No.	Functional Requirement	Test Steps	Expected Result	Passed/Failed
1	Users have an ability to open Reference Data Factory (RDF) application	Open RDF	Application opened	Passed
2	Users have an ability to open folder "Instruments" and open search page	Go to Menu Go to folder "Instruments" Click to View	Page "View" successfully opened and functional	Passed
3	Users should have a possibility to interact with "View" page	Put following data in RDF: ISIN code == Code that was provided by Customer Instrument Status == "ACTIVE" CBF/CBL/LUXSD Holding Flags == NO	All data correctly setup in "View" page	Passed
4	RDF must start search data by user's request	Click to "Search" button	Successfully searched and result page appeared	Passed
5	RDF must reflect on user's request	Click to "Mature Instrument" button	Instrument Details page is successfully opened	Passed
6	RDF must check and validate correctness	Once Instrument Details opens -> press Default & Validate button	RDF should populate "Validation is successful"	Passed
7	Once changes were successfully succeeded system will populate message.	Click to "Save and Close" button	RDF must reflect following message "Changes applied for the ISIN…"	Passed

Source: own processing

Table 3: Test Cases for 2nd Requirement

Test Requirement 2 – Re-Open significant number of financial instruments				
No.	Functional Requirement	Test Steps	Expected Result	Passed/Failed
1	Users have an ability to open Reference Data Factory (RDF) application	Open RDF	Application opened	Passed
2	Users have an ability to open folder "Instruments" and open search page	Go to Menu Go to folder "Instruments" Click to View	Page "View" successfully opened and functional	Passed
3	Users should have a possibility to interact with "View" page	Put following data in RDF: ISIN code == Code that was provided by Customer Instrument Status == "MATURED" CBF/CBL/LUXSD Holding Flags == NO	All data correctly setup in "View" page	Passed
4	RDF must start search data by user's request	Click to "Search" button	Successfully searched and result page appeared	Passed
5	RDF must reflect on user's request	Click to "Re-open Instrument" button	Instrument Details page is successfully opened	Passed
6	RDF must check and validate correctness	Once Instrument Details opens -> press Default & Validate button	RDF should populate "Validation is successful"	Passed
7	Once changes were successfully succeeded system will populate message.	Click to "Save and Close" button	RDF must reflect following message "Changes applied for the ISIN…"	Passed

Source: own processing

Table 4: Test Cases for 3rd Requirement

Test Requirement 3 – Cancel significant number of financial instruments				
No.	Functional Requirement	Test Steps	Expected Result	Passed/Failed
1	Users have an ability to open Reference Data Factory (RDF) application	Open RDF	Application opened	Passed
2	Users have an ability to open folder "Instruments" and open search page	Go to Menu Go to folder "Instruments" Click to View	Page "View" successfully opened and functional	Passed
3	Users should have a possibility to interact with "View" page	Put following data in RDF: ISIN code == Code that was provided by Customer Instrument Status == "ACTIVE" CBF/CBL/LUXSD Holding Flags == NO	All data correctly setup in "View" page	Passed
4	RDF must start search data by user's request	Click to "Search" button	Successfully searched and result page appeared	Passed
5	RDF must reflect on user's request	Click to "Cancel Instrument" button	Instrument Details page is successfully opened	Passed
6	RDF must check and validate correctness	Once Instrument Details opens -> press Default & Validate button	RDF should populate "Validation is successful"	Passed
7	Once changes were successfully succeeded system will populate message.	Click to "Save and Close" button	RDF must reflect following message "Changes applied for the ISIN…"	Passed

Source: own processing

Table 5: Test Cases for 4th Requirement

Test Requirement 4 – Reactivate significant number of financial instruments				
No.	Functional Requirement	Test Steps	Expected Result	Passed/Failed
1	Users have an ability to open Reference Data Factory (RDF) application	Open RDF	Application opened	Passed
2	Users have an ability to open folder "Instruments" and open search page	Go to Menu Go to folder "Instruments" Click to View	Page "View" successfully opened and functional	Passed
3	Users should have a possibility to interact with "View" page	Put following data in RDF: ISIN code == Code that was provided by Customer Instrument Status == "CANCELLED" CBF/CBL/LUXSD Holding Flags == NO	All data correctly setup in "View" page	Passed
4	RDF must start search data by user's request	Click to "Search" button	Successfully searched and result page appeared	Passed
5	RDF must reflect on user's request	Click to "Re-activate Instrument" button	Instrument Details page is successfully opened	Passed
6	RDF must check and validate correctness	Once Instrument Details opens -> press Default & Validate button	RDF should populate "Validation is successful"	Passed
7	Once changes were successfully succeeded system will populate message.	Click to "Save and Close" button	RDF must reflect following message "Changes applied for the ISIN…"	Passed

Source: own processing

4.3 Manual Testing

This chapter details the manual testing conducted on selected requirements of the web application. The testing involved the participation of three testers, each adhering to pre-established test cases. The testing transpired across three distinct phases with a meticulous recording of the time required for the test cases to successfully pass. The time recording procedure followed a sequential testing approach, wherein the tester consecutively assessed the first, second, and third test cycles. Subsequently, the cumulative time for the entire cycle was computed by summing up these individual values. Following the completion of the recording process, a tabular represented the recorded result. To derive insights into the temporal aspects of testing, an arithmetic mean was computed based on the recorded values.

Table 6: Manual Testing Results – 1st Tester

Test Cases	Tester #1 (Junior) 1 ISIN	Tester #1 (Junior) 5 ISIN	Tester #1 (Junior) 20 ISIN
Test Case 1	2m 21s	11m 55s	46m 27s
Test Case 2	2m 1s	11m 44s	47m 5s
Test Case 3	2m 15s	12m 2s	50m 0s
Test Case 4	2m 6s	11m 55s	45m 11s
1st Cycle Results	8m 43s	47m 36s	3h 8m 43s
Test Case 1	2m 0s	10m 51s	44m 22s
Test Case 2	2m 1s	10m 44s	43m 51s
Test Case 3	2m 10s	11m 28s	47m 10s
Test Case 4	2m 0s	11m 55s	47m 51s
2nd Cycle Results	8m 43s	44m 58s	3h 3m 14s
Test Case 1	2m 5s	10m 53s	46m 21s
Test Case 2	2m 8s	11m 4s	45m 25s
Test Case 3	2m 0s	10m 52s	43m 10s
Test Case 4	2m 6s	12m 0s	47m 15s
3rd Cycle Results	8m 43s	44m 49s	3h 2m 6s

Source: own processing

Table 7: Manual Testing Results – 2nd Tester

Test Cases	Tester #2 (Middle) 1 ISIN	Tester #2 (Middle) 5 ISIN	Tester #2 (Middle) 20 ISIN
Test Case 1	2m 0s	11m 0s	45m 0s
Test Case 2	2m 0s	11m 0s	47m 0s
Test Case 3	2m 15s	12m 0s	47m 0s
Test Case 4	2m 0s	11m 0s	45m 0s
1st Cycle Results	8m 43s	47m 36s	3h 8m 43s
Test Case 1	2m 0s	10m 50s	45m 0s
Test Case 2	2m 0s	10m 50s	44m 0s
Test Case 3	2m 10s	11m 30s	47m 0s
Test Case 4	2m 0s	11m 0s	47m 0s
2nd Cycle Results	8m 43s	44m 58s	3h 3m 14s
Test Case 1	2m 0s	11m 0s	45m 0s
Test Case 2	2m 0s	11m 0s	45m 30s
Test Case 3	2m 0s	11m 0s	43m 0s
Test Case 4	2m 0s	12m 0s	47m 0s
3rd Cycle Results	8m 43s	44m 49s	3h 2m 6s

Source: own processing

Table 8: Manual Testing Results – 3rd Tester

Test Cases	Tester #3 (Junior) 1 ISIN	Tester #3 (Junior) 5 ISIN	Tester #3 (Junior) 20 ISIN
Test Case 1	2m 51s	13m 34s	50m 27s
Test Case 2	2m 31s	13m 48s	53m 5s
Test Case 3	2m 59s	13m 29s	58m 0s
Test Case 4	2m 54s	13m 55s	55m 11s
1st Cycle Results	11m 15s	54m 46s	3h 36m 43s
Test Case 1	2m 40s	13m 51s	54m 22s
Test Case 2	3m 1s	14m 44s	50m 51s
Test Case 3	3m 10s	14m 28s	58m 10s
Test Case 4	2m 50s	13m 55s	57m 51s
2nd Cycle Results	11m 41s	56m 58s	3h 41m 14s
Test Case 1	2m 52s	12m 53s	49m 21s
Test Case 2	3m 38s	13m 4s	49m 25s
Test Case 3	2m 44s	14m 52s	51m 10s
Test Case 4	2m 56s	14m 0s	54m 15s
3rd Cycle Results	12m 0s	54m 49s	3h 24m 11s

Source: own processing

This meticulous recording and analysis process allows for a detailed understanding of the temporal dynamics associated with the testing phases.

4.4 Automated Testing with Python, Selenium and Jenkins tools

Initially, it is essential to set up the following software and prerequisites on a computer running a supported operating system to generate automated test cases:

1) Python
2) Selenium WebDriver
3) Gherkin plugins
4) Jenkins
5) Python Behave Framework
6) IDE

Python is a high-level programming language known for its simplicity and readability of code. Widely used in various fields including web application development, scientific research, data analysis and test automation.

Thesis Application: You may consider using Python to create automated tests using Selenium WebDriver and Behave Framework, as well as to process data and analyze test results.

Selenium WebDriver is a powerful tool for automating the testing of web applications. Allows you to programmatically interact with browsers, perform user actions and check the status of web pages.

Thesis Application: Consider using Selenium WebDriver to create automated tests of web applications and analyze their performance and reliability.

Gherkin is a human-readable language for writing scripts for system behavior. Gherkin-related plugins can provide additional functionality and integrations for using the language in different development environments.

Thesis Application: Explore plugins that extend the functionality of Gherkin and consider their impact on the efficiency of writing and running automated tests.

Jenkins is a continuous integration server designed to automate the build, test, and deployment of software.

Thesis Application: Consider using Jenkins to create a continuous integration and delivery process in the context of automated testing.

Behave is a framework for writing automated tests that use Gherkin to define scripts and Python to implement them.

Thesis Application: Explore the use of Behave in combination with Python to create structured and understandable auto-tests based on behavioral scenarios.

IDE (Integrated Development Environment) is an integrated development environment that provides tools and functionality for working effectively with code.

Thesis Application: Consider the selection and use of IDEs when developing, debugging, and maintaining automated test code and scripts.

4.4.1 Gherkin Language Scenarios, BDD test

The print screens below showcase the primary steps of execution of all test requirements. Following the principles of behavior-based design (BDD) scenarios, structured sequentially, provides a step-by-step navigation guide for testers. Through the application, interact with the user interface, and check the expected results. The scenarios cover a range of instrument-related functions, providing a thorough examination of the software's behavior.

```
@normal @UI @MatureInstrument
  Scenario Outline: Mature instrument

    Given I go to "RDF" site
    When I click on "Instruments" link
    And I click on "View" link
    And I type "<Isin Code>" in "Isin Code" field
    And I type "<Instrument status>" in "Instrument status" field
    And I click on "Advanced Search criteria 1" tab
    And I select "<CBL/LuxCSD Holding Indicator>" from the "CBL/LuxCSD
Holding Indicator" dropdown
    And I select "<CBF Holding Indicator>" from the "CBF Holding Indicator"
dropdown
    And I select "<CBL ICSD Holding Indicator>" from the "CBL ICSD Holding
Indicator" dropdown
    And I select "<LuxCSD ICSD Holding Indicator>" from the "LuxCSD ICSD
Holding Indicator" dropdown
```

```
    And I select "<CBF T2S Holding Indicator>" from the "CBF T2S Holding
Indicator" dropdown
    And I select "<LuxCSD T2S Holding Indicator>" from the "LuxCSD T2S
Holding Indicator" dropdown
    And I click on "Search" button
    And I wait for "10" seconds
    And I click on "<Mature Instrument>" button
    And I click on "Default And Validate" button
    And I validate "Message" contains "Validation Successful"
    And I click on "Save And Close" button
    #reason to be added for Microsoft Explorer
    Then I validate "Message" contains "Changes applied for instrument"
```

```
  @normal @UI @ReactivateInstrument
  Scenario Outline: Reactivate instrument

    Given I go to "RDF" site
    When I click on "Instruments" link
    And I click on "View" link
    And I type "<Isin Code>" in "Isin Code" field
    And I type "<Instrument status>" in "Instrument status" field
    And I click on "Search" button
    And I wait for "10" seconds
    And I click on "<Reactivate Instrument>" button
    And I click on "Default And Validate" button
    And I validate "Message" contains "Validation Successful"
    And I click on "Save And Close" button
    #reason to be added in ST env
    #potential pending duplicate
    Then I validate "Message" contains "Reactivation done for instrument"
```

```
@normal @UI @CancelInstrument
  Scenario Outline: Cancel instrument

    Given I go to "RDF" site
    When I click on "Instruments" link
    And I click on "View" link
    And I type "<Isin Code>" in "Isin Code" field
    And I type "<Instrument status>" in "Instrument status" field
    And I click on "Advanced Search criteria 1" tab
    And I select "<CBL/LuxCSD Holding Indicator>" from the "CBL/LuxCSD
Holding Indicator" dropdown
    And I select "<CBF Holding Indicator>" from the "CBF Holding Indicator"
dropdown
```

```
    And I select "<CBL ICSD Holding Indicator>" from the "CBL ICSD Holding
Indicator" dropdown
    And I select "<LuxCSD ICSD Holding Indicator>" from the "LuxCSD ICSD
Holding Indicator" dropdown
    And I select "<CBF T2S Holding Indicator>" from the "CBF T2S Holding
Indicator" dropdown
    And I select "<LuxCSD T2S Holding Indicator>" from the "LuxCSD T2S
Holding Indicator" dropdown
    And I click on "Search" button
    And I wait for "10" seconds
    And I click on "<Cancel Instrument>" button
    And I click on "Default And Validate" button
    And I validate "Message" contains "Validation Successful"
    And I click on "Save And Close" button
    #reason to be added for Microsoft Explorer
    Then I validate "Message" contains "Cancellation done for instrument"
```

```
@normal @UI @ReopenInstrument
  Scenario Outline: Re-Open instrument

    Given I go to "RDF" site
    When I click on "Instruments" link
    And I click on "View" link
    And I type "<Isin Code>" in "Isin Code" field
    And I type "<Instrument status>" in "Instrument status" field
    And I click on "Search" button
    And I wait for "10" seconds
    And I click on "<Reopen Instrument>" button
    And I type "<Initial maturity date>" in "Initial maturity date" field
    And I click on "Default And Validate" button
    And I validate "Message" contains "Validation Successful"
    And I click on "Save And Close" button
    #reason to be added in ST env
    #potential pending duplicate
    Then I validate "Message" contains "Reopen done for instrument"
```

Furthermore, the visual aid provided by the print screens enhances the clarity and accessibility of the testing process. Testers can easily reference the step-by-step guide, fostering efficiency and accuracy in their execution.

Incorporating BDD principles into the testing process not only improves communication between stakeholders but also promotes a shared understanding of the software's

requirements and expected behavior. This collaborative approach minimizes the risk of misunderstandings and enhances the overall quality of the software being tested.

4.4.2 Test Execution and Automation Outputs

Jenkins is a prior tool to execute newly added scenarios. Main advantage of Jenkins is its seamless integration with Allure, a reporting tool that offers comprehensive insights into test results. Upon successful execution of test cases, Jenkins can generate detailed Allure Reports, providing valuable information for testers. The report displays a list of successfully passed tests in the Behaviors tab, offering a clear overview of the testing outcomes. Additionally, the report includes execution time data, a crucial metric that will later serve for comparing results with those obtained through manual testing.

In essence, Jenkins, coupled with Allure, streamlines the testing process by not only automating test execution but also furnishing detailed reports. These reports become invaluable tools for testers, offering visibility into passed tests, execution times, and facilitating comparisons with manual testing results. The seamless integration between Jenkins and Allure enhances the efficiency and effectiveness of the overall testing workflow.

Snippets below show the execution of all test requirements for a single ISIN code in Jenkins.

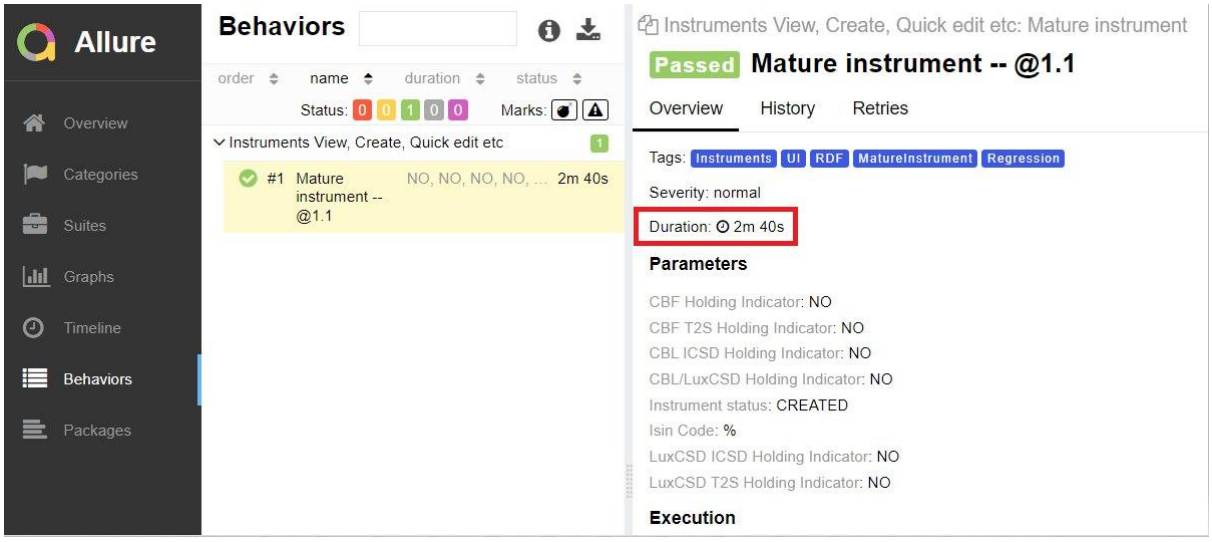

Figure 7: Execution Output for 1st requirement, Source: own processing

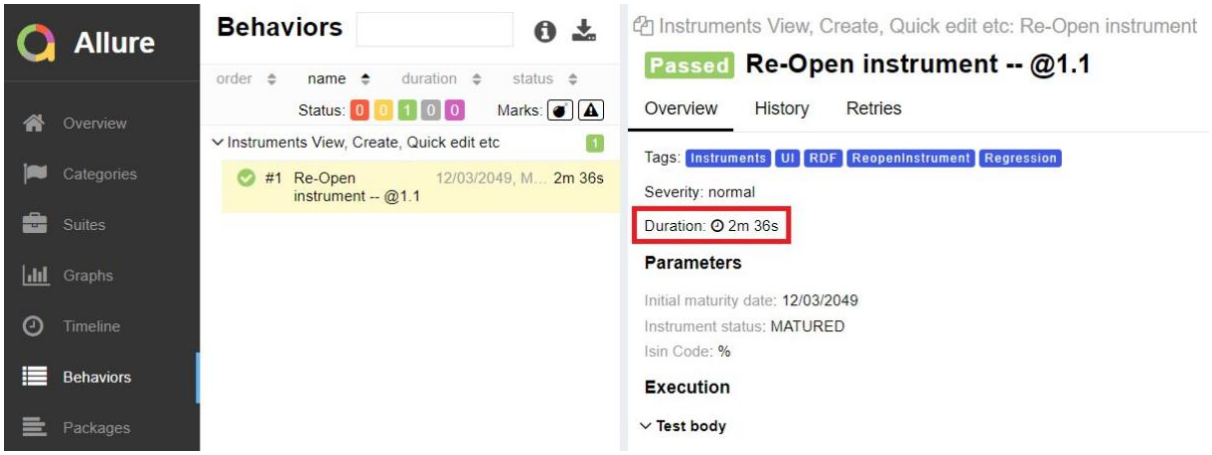

Figure 8: Execution Output for 2nd requirement, Source: own processing

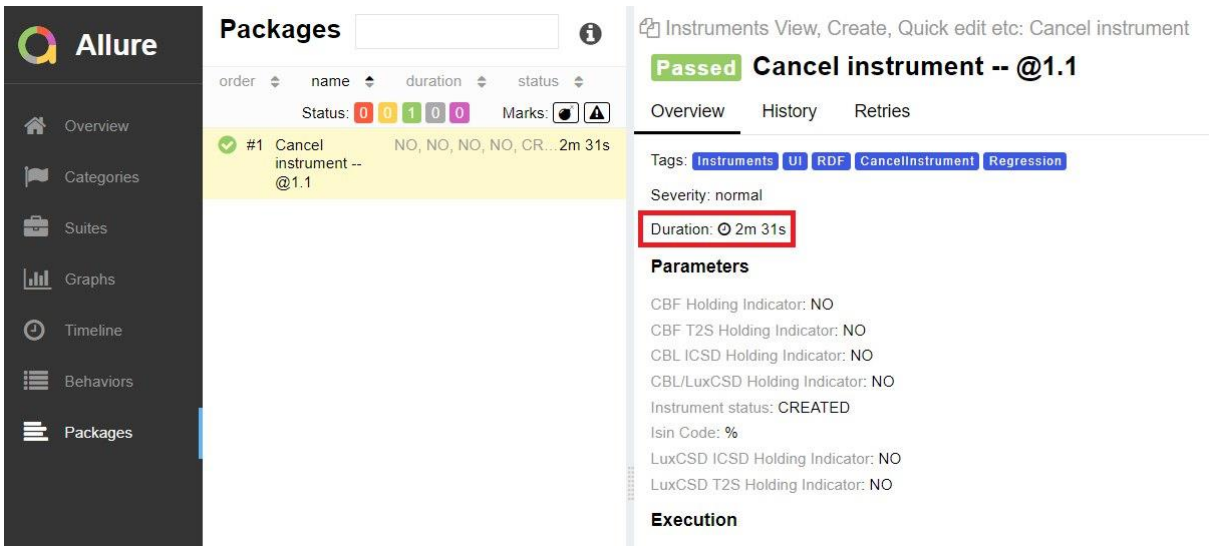

Figure 9: Execution Output for 3rd requirement, Source: own processing

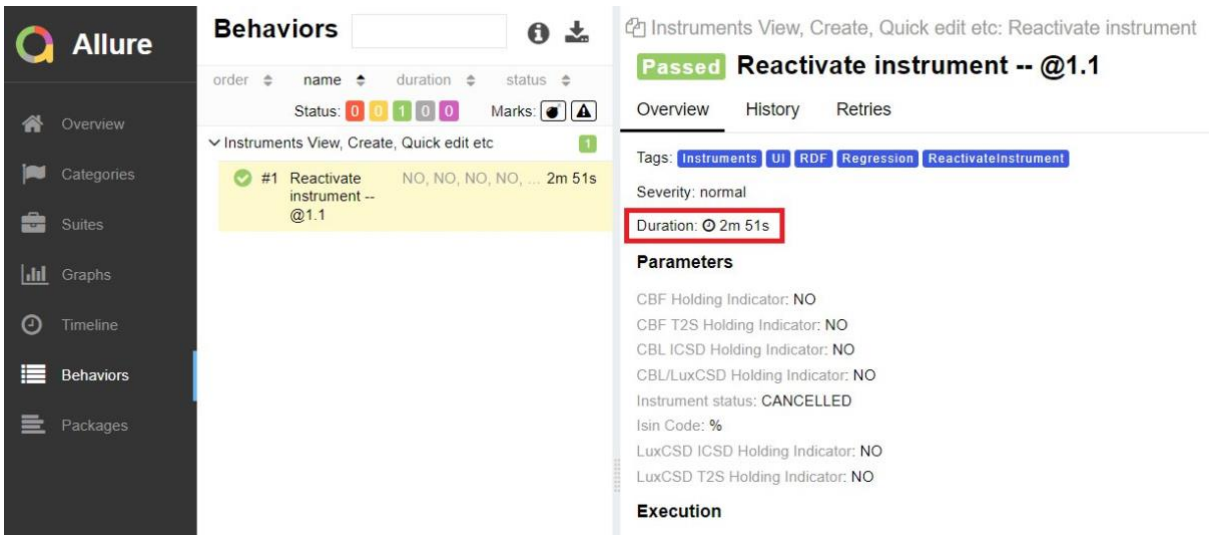

Figure 10:Execution Output for 4th requirement, Source: own processing

The following results highlight the time efficiency achieved through the automation of 1, 5, and 20 ISIN (International Securities Identification Number) scenarios.

Table 9: Automated Testing Result

Test Cases	Tester #3 (Junior) 1 ISIN	Tester #3 (Junior) 5 ISIN	Tester #3 (Junior) 20 ISIN
Test Case 1	2m 40s	10m 33s	41m 33s
Test Case 2	2m 36s	10m 58s	42m 36s
Test Case 3	2m 31s	11m 0s	41m 0s
Test Case 4	2m 51s	10m 25s	41m 16s
1st Cycle Results	10m 38s	42m 56s	2h 46m 25s
Test Case 1	2m 10s	11m 23s	41m 33s
Test Case 2	2m 5s	10m 41s	41m 13s
Test Case 3	2m 16s	10m 33s	40m 0s
Test Case 4	2m 37s	10m 8s	40m 10s
2nd Cycle Results	9m 8s	42m 45s	2h 42m 56s
Test Case 1	2m 33s	10m 6s	42m 33s
Test Case 2	2m 19s	10m 46s	41m 26s
Test Case 3	2m 24s	10m 26s	41m 0s
Test Case 4	2m 44s	11m 2s	41m 10s
3rd Cycle Results	9m 35s	42m 20s	2h 46m 9s

Source: own processing

5 Results and Discussion

Upon concluding the testing phase and obtaining results, it becomes feasible to assess and contrast the difference between manual and automated testing methods applied to used web application. The evaluation will be conducted based on criteria that are crucial for all parties involved, including:

- Time and Efficiency
- Usability Factors

5.1 Time and Efficiency

Having measured and acquired the execution speed of each test case in the preceding chapters, the focus will now shift to a straightforward comparison. In the earlier sections of this thesis, three average values were derived for manual testing, while a singular average time value was obtained for automated testing. Additionally, it is crucial to establish an average time for manual testing to facilitate a comprehensive comparison. The objective is to calculate the ratio of test execution time between the two testing methods and establish efficiency.

Table 10: Test Comparison - 1 ISIN

Test Comparison - Time & Efficiency - 1 ISIN		
Tester #1 (Junior)	8m 12s	
Tester #2 (Junior)	8m 8s	
Tester #1 (Junior)	12m 4s	Automation
Average	9m 46s	9m 25s
Ratio	1: 1,0546	

Source: own processing

Table 11: Test Comparison - 5 ISIN

Test Comparison - Time & Efficiency - 5 ISIN		
Tester #1 (Junior)	46m 19s	
Tester #2 (Junior)	45m 0s	
Tester #1 (Junior)	55m 30s	Automation
Average	48m 56s	42m 40s
Ratio	1: 0,871	

Source: own processing

Table 12: Test Comparison - 20 ISIN

Test Comparison - Time & Efficiency – 20 ISIN		
Tester #1 (Junior)	3h 4m 40s	Automation
Tester #2 (Junior)	3h 2m 30s	
Tester #1 (Junior)	3h 12m 23s	
Average	3h 6m 31s	2h 44m 30s
Ratio	1: 0,881	

Source: own processing

The findings highlight a consistent trend of time savings with test automation across different test case volumes. The efficiency ratios emphasize that, on average, automated testing outperforms manual testing by approximately 8.7% to 10.5%, showcasing its superiority in terms of execution speed and overall efficiency.

5.2 Usability Factors

Usability, as reflected in the time savings and efficiency ratios, emerges as a strong suit for automated testing. The ratios (ranging from 1:0.871 to 1:1.0546) signify a user-friendly integration of automated testing into the testing process. This ease of use contributes to quicker and more consistent test executions, making automated testing accessible to testing teams with varying levels of experience.

As your app grows, the tests for it need to change too—whether they're done manually or automatically. If a change is made in the app and it's covered by test scenarios, some tests may fail because they're no longer a good match. Even a small tweak in the code order can make one or more tests not work as expected. The adaptability of automated testing shines through in its consistent outperformance in terms of efficiency across test case volumes. It is important for changes in the software, addressing regression testing needs, and maintaining reliability across different software releases.

The analysis paints a compelling picture of automated testing as a superior choice over manual regression testing in terms of usability, lower resource impact, flexibility, and adaptability. These advantages not only translate into tangible time savings but also underscore the viability of automated testing in dynamic development environments where software is subject to constant evolution.

5.3 Discussion

Research Question 1: *Considering the comparison between the two distinct testing approaches, which one stands out as more effective for testing a complex system?*

The primary discovery in this research indicates that automation offers various benefits rather than a manual approach for testing complex systems. Automated testing has emerged as a powerhouse for handling repetitive tasks and ensuring the consistent execution of many test cases. Certain types of tests, especially simpler standard operations, could surely be automated. Nevertheless, it's critical to note that repeatedly running the same automated test, without any code modifications, may lead to occasional test failures, making it unreliable.

Research Question 2: *Is it possible for the company to completely transition from manual regression testing to automated testing for the selected web application?*

Initially, the expectation in this research was that automated testing held the solution. Automating processes seemed logical to enhance tester efficiency. However, the realization after the research and discussion is that automated tests shouldn't be viewed to fully replace human testing but to support and extend test reach.

In conclusion, this study suggests that a combination of manual testing and the power of automation is the most suitable strategy for testing complex systems such as RDF. Despite the benefits of test automation, the human factor and capabilities cannot be excluded from the testing process. Therefore, complete automation of a complex system like RDF may not be optimal.

6 Conclusion

This thesis opens with a thorough exploration of testing concepts and their primary classifications through an extensive literature review. The theoretical part includes an examination of the advantages and disadvantages of various Software Development Life Cycle (SDLC) models, with detailed explanations of each stage. The diverse types of tests prevalent in the modern world are also studied and outlined in this section.

The practical part involves the testing of a selected application using automated and manual regression testing methods. The theoretical knowledge acquired is applied to the practical part of the thesis. The purpose is to compare automated and manual testing for a web application designed for a stock exchange firm. This section provides a detailed description of the tested application and the chosen test automation tools. Test cases are formulated based on the application's requirements, and three testers manually execute these cases, recording the time taken in a table.

Upon obtaining the manual testing results, test steps are translated into the Gherkin language, and main functions are scripted in Python for test automation. These scripts are integrated into the Jenkins application, generating a report through Allure. The automated test results are compiled into a table for analysis. The comparison focuses on the following parameters: time, efficiency, and usability.

In the end, the findings of this study emphasize the significance of adopting a balanced approach to testing complex systems, particularly in the context of web applications. The research underscores the effectiveness of combining manual testing with the capabilities of automation to create a strong testing strategy. While automation brings undeniable advantages, this study highlights the crucial role of human involvement in the testing process. The synthesis of human expertise and automation power emerges as a compelling strategy for achieving comprehensive and reliable testing results in intricate systems.

7 References

FORGÁCS, István and KOVÁCS, Attila, 2023. *Modern software testing techniques: A Practical Guide for Developers and Testers*. Apress. 266 p. ISBN-13: 9781484298923

LOVELAND, Scott, MILLER, Geoffrey, PREWITT, Richard, Jr and SHANNON, Michael, 2013. *Software Testing Techniques: Finding the Defects that Matter*. 386 p. ISBN-13: 9781627040235

KINSBRUNER, Eran, 2022. *A frontend web Developer's guide to testing: Explore Leading Web Test Automation Frameworks and Their Future Driven by Low-code and AI*. 304 p. ISBN-13: 9781803238319

HIGHTOWER, Richard and LESIECKI, Nicholas, 2001. *Java tools for extreme programming: Mastering Open Source Tools, Including Ant, JUnit, and Cactus*. Wiley. 544 p. ISBN-13: 9780471207085

GULATI, Shekhar and SHARMA, Rahul, 2017. *Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5*. Apress. 151 p. ISBN-13: 9781484230145

FEWSTER, Mark and GRAHAM, Dorothy, 1999. *Software test automation: Effective Use of Test Execution Tools*. Addison-Wesley Professional. 600 p. ISBN-13: 9780201331400 –

GRAHAM, Dorothy and FEWSTER, Mark, 2012. *Experiences of test automation: Case Studies of Software Test Automation*. Addison-Wesley Professional. 672 p. ISBN-13: 9780321754066

SAMBAMURTHY, Manikandan, 2023. *Test Automation Engineering Handbook: Learn and Implement Techniques for Building Robust Test Automation Frameworks*. Packt Publishing. 276 p. ISBN-13: 9781804615492

LEWIS, William E., 2008. *Software Testing and Continuous Quality Improvement, third edition*. Auerbach Publications. 684 p. ISBN-13: 9781420080735

EVERETT, Gerald D. and MCLEOD, Raymond, Jr, 2007. *Software testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Press. 280 p. ISBN-13: 9780471793717

LELOUDAS, Panagiotis, 2023. *Introduction to software testing: A Practical Guide to Testing, Design, Automation, and Execution*. Apress. 211 p. ISBN-13: 9781484295137

BIERIG, Ralf, BROWN, Stephen, GALVÁN, Edgar and TIMONEY, Joe, 2021. *Essentials of software testing*. Cambridge University Press. 318 p. ISBN-13: 9781108833349

HOMÈS, Bernard, 2012. *Fundamentals of software testing*. Wiley-ISTE. 384 p. ISBN-13: 9781848213241

JOSE, 2021. *Test automation: A Practitioner's Guid*. BCS, The Chartered Institute for IT. 274 p. ISBN-13: 9781780175454

HUNT, Andrew, THOMAS, Dave and HARGETT, Matt, 2007. *Pragmatic Unit Testing in C# with NUnit*. 239 p. ISBN-13: 9780977616671

KAUL, Neha, 2022. *Implementing automated software testing*. Arcler Press. 277 p. ISBN-13: 9781774694039

ALPAEV, Gennadiy, 2017. *Software testing automation tips: 50 Things Automation Engineers Should Know*. Apress. 50 p. ISBN-13: 9781484231616

TUDOSE, Catalin, 2020. *JUnit in Action, third edition*. Manning Publications. 525 p. ISBN-13: 9781617297045

8 List of pictures, tables, graphs, and abbreviations

8.1 List of figures

8.2 List of tables

9 Appendix

Python code

```
  from tabulate import tabulate
import ast
from deepdiff import DeepDiff
from selenium import webdriver
# from selenium.webdriver import DesiredCapabilities
from selenium.webdriver.chrome.options import Options as ChromeOptions
from selenium.common.exceptions import StaleElementReferenceException
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from BDDCommon.CommonConfigs import urlconfig
import time
from time import sleep
import logging as logger
import allure
from allure_commons.types import AttachmentType
from datetime import datetime
from selenium.webdriver.support.ui import Select
from BDDCommon.CommonConfigs import locatorsconfig
import os
import pathlib
import json
import pandas as pd
from selenium.webdriver.common.keys import Keys

def go_to(context, location, **kwargs):
    with allure.step("Opening Browser"):
        logger.info("The detailed report of the scenario is present
here:\n{}\n\n".format(
            context.config.userdata.get('allureurl')))
        browser = context.config.userdata.get('browser')
        env = context.config.userdata.get('environment')
        grid = context.config.userdata.get('grid')
        video = context.config.userdata.get('video')
        certificate = context.config.userdata.get('certificate')
        username = context.config.userdata.get('username').replace("_", " ")
        root_dir = pathlib.Path(__file__).parents[2].absolute()

        if not browser:
            browser = 'firefox'
        if not grid:
            grid = 'y'
```

```
        try:
            url = urlconfig.URL[env.lower()][location.lower()]
        except Exception as e:
            raise Exception("The environment for '{}' is not
supported".format(env))
        if grid.lower() in ('y', 'yes'):
            cloud_options = {}
            if browser.lower() == 'chrome':
                options = ChromeOptions()
                cloud_options['browserName'] = 'chrome'
                cloud_options['javascriptEnabled'] = True
                options.set_capability('cloud:options', cloud_options)
                # caps = DesiredCapabilities.CHROME.copy()
                # caps['browserName'] = 'chrome'
                # caps['javascriptEnabled'] = True
            elif browser.lower() in ('ff', 'firefox'):
                # caps = DesiredCapabilities.FIREFOX.copy()
                # caps['browserName'] = 'firefox'
                # caps['javascriptEnabled'] = True
                # caps['acceptInsecureCerts'] = True
                cloud_options['browserName'] = 'firefox'
                cloud_options['javascriptEnabled'] = True
                cloud_options['acceptInsecureCerts'] = True
                profile_path = os.path.join(root_dir, 'profiles', 'firefox',
certificate)
                if not os.path.exists(profile_path):
                    assert False, "Profile for user {} of {} does not
exist".format(certificate, username)
                options = webdriver.FirefoxOptions()
                profile =
webdriver.FirefoxProfile(profile_directory=profile_path)
                profile.set_preference("security.default_personal_cert",
"Select Automatically")
                profile.set_preference("security.osclientcerts.autoload",
True)
                profile.set_preference("layout.css.devPixelsPerPx", "0.9")
                #
profile.set_preference("security.disable_button.openCertManager", True)
                #
profile.set_preference("security.enterprise_roots.enabled", True)
                # profile.set_preference("accept_untrusted_certs", True)
                # profile.set_preference("assume_untrusted_cert_issuer",
True)
                options.profile = profile
                options.log.level = "trace"
                # options.headless = True
                if video.lower() == 'yes':
                    options.set_capability("se:recordVideo", True)
```

```
                    options.set_capability("se:timeZone",
"Europe/Luxembourg")
                    options.set_capability("se:screenResolution",
"1920x1080")
                options.set_capability('cloud:options', cloud_options)
            else:
                raise Exception("The browser type '{}' is not
supported".format(browser))

            try:
                # context.driver =
webdriver.Remote(desired_capabilities=caps,
command_executor='http://rdhsselen01.deutsche-boerse.de/wd/hub', options =
options)
                context.driver =
webdriver.Remote(command_executor='http://rdhsselen01.deutsche-
boerse.de/wd/hub',
                                                    options=options)
                context.driver.maximize_window()
                context.temp_dir_name = max(
                    (os.path.join("/tmp", d) for d in os.listdir("/tmp") if
d.startswith("tmp")),
                    key=os.path.getctime,
                    default=None
                )
            except Exception as e:
                print(e)
        else:
            if browser.lower() == 'chrome':
                options = webdriver.ChromeOptions()
                options.add_argument("--disable-infobars")
                options.add_argument("--disable-notifications")
                options.add_argument("--start-maximized")
                options.add_experimental_option('excludeSwitches', ['enable-
logging'])
                context.driver = webdriver.Chrome(options=options)
            elif browser.lower() == 'headlesschrome':
                options = webdriver.ChromeOptions()
                options.add_argument('--headless')
                context.driver = webdriver.Chrome(options=options)
            elif browser.lower() in ('ff', 'firefox'):
                profile_path = os.path.join(root_dir, 'profiles', 'firefox',
'mple6cvr.SeleniumNode')
                options = webdriver.FirefoxOptions()
                profile =
webdriver.FirefoxProfile(profile_directory=profile_path)
                profile.set_preference("security.default_personal_cert",
"Select Automatically")
```

```
                profile.set_preference("security.osclientcerts.autoload",
True)
                profile.set_preference("layout.css.devPixelsPerPx", "0.9")
                options.profile = profile
                options.log.level = "trace"
                context.driver = webdriver.Firefox(options=options)
            elif browser.lower() == 'edge':
                context.driver = webdriver.Edge()
            else:
                raise Exception("The browser type '{}' is not
supported".format(browser))
        context.driver.maximize_window()
        wait = int(kwargs['implicitly_wait']) if 'implicitly_wait' in
kwargs.keys() else 10
        context.driver.implicitly_wait(wait)

        url = url.strip()
        context.driver.get(url)
        context.driver.set_window_size(1920, 1080)
        TakeScreenShot(context)
```

```
def ReportFailure(context, message):
    with allure.step("Reporting Failure"):
        allure.attach(context.driver.get_screenshot_as_png(), "Screenshot-"
+ datetime.now().strftime('%Y%m%d%H%M%S'),
                      AttachmentType.PNG)
        logger.error(message)
        allure.attach(message, "ErrorMessage-" +
datetime.now().strftime('%Y%m%d%H%M%S'), AttachmentType.TEXT)
        assert False, message
```

```
def ReportFailure(context, message):
    with allure.step("Reporting Failure"):
        allure.attach(context.driver.get_screenshot_as_png(), "Screenshot-"
+ datetime.now().strftime('%Y%m%d%H%M%S'),
                      AttachmentType.PNG)
        logger.error(message)
        allure.attach(message, "ErrorMessage-" +
datetime.now().strftime('%Y%m%d%H%M%S'), AttachmentType.TEXT)
        assert False, message
```

```
def TakeScreenShot(context):
    allure.attach(context.driver.get_screenshot_as_png(), "Screenshot-" +
datetime.now().strftime('%Y%m%d%H%M%S'),
                  AttachmentType.PNG) if
context.config.userdata.get('screenshot').lower() == "yes" else None
```

Log files

```
Started by user Bekzat Kelgenbayev
Obtained TestAutomationBDDFramework/jenkins/EndToEndTestRDF
[Pipeline] Start of Pipeline
[Pipeline] node
Running on TestAutomation-sim-onprem in /svc/selenium/jenkins-
agent/workspace/testing/RDF/End_To_End_Testing
+ source /svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin/activate
++ deactivate nondestructive
++ '[' -n '' ']'
++ '[' -n '' ']'
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
++ '[' -n '' ']'
++ unset VIRTUAL_ENV
++ '[' '!' nondestructive = nondestructive ']'
++ VIRTUAL_ENV=/svc/selenium/jenkins-agent/workspace/testing/venv_rdh
++ export VIRTUAL_ENV
++ _OLD_VIRTUAL_PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
++ PATH=/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin
++ export PATH
++ '[' -n '' ']'
++ '[' -z '' ']'
++ _OLD_VIRTUAL_PS1=
++ '[' 'x(venv_rdh) ' '!=' x ']'
++ PS1='(venv_rdh) '
++ export PS1
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
+ cd TestAutomationBDDFramework
+ python3 -m runner --job_dir testing/RDF/End_To_End_Testing --run_allure
true '--behave_options=-t Regression -t MatureInstrument -D
environment=June2022-BAT -D browser=Firefox -D allstepexecution=No -D
video=Yes -D screenshot=Yes'
/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/lib64/python3.6/site-
packages/azure/storage/blob/_shared/encryption.py:15:
CryptographyDeprecationWarning: Python 3.6 is no longer supported by the
Python core team. Therefore, support for it is deprecated in cryptography
and will be removed in a future release.
  from cryptography.hazmat.backends import default_backend
1 feature passed, 0 failed, 23 skipped
1 scenario passed, 0 failed, 19 skipped
19 steps passed, 0 failed, 431 skipped, 0 undefined
Took 2m31.278s
Allure report was successfully generated.
```

```
Creating artifact for the build.
Artifact was added to the build.
Finished: SUCCESS
```

```
Started by user Bekzat Kelgenbayev
Obtained TestAutomationBDDFramework/jenkins/EndToEndTestRDF
[Pipeline] Start of Pipeline
[Pipeline] node
Running on TestAutomation-sim-onprem in /svc/selenium/jenkins-
agent/workspace/testing/RDF/End_To_End_Testing
+ source /svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin/activate
++ deactivate nondestructive
++ '[' -n '' ']'
++ '[' -n '' ']'
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
++ '[' -n '' ']'
++ unset VIRTUAL_ENV
++ '[' '!' nondestructive = nondestructive ']'
++ VIRTUAL_ENV=/svc/selenium/jenkins-agent/workspace/testing/venv_rdh
++ export VIRTUAL_ENV
++ _OLD_VIRTUAL_PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
++ PATH=/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin
++ export PATH
++ '[' -n '' ']'
++ '[' -z '' ']'
++ _OLD_VIRTUAL_PS1=
++ '[' 'x(venv_rdh) ' '!=' x ']'
++ PS1='(venv_rdh) '
++ export PS1
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
+ cd TestAutomationBDDFramework
+ python3 -m runner --job_dir testing/RDF/End_To_End_Testing --run_allure
true '--behave_options=-t Regression -t ReopenInstrument -D
environment=June2022-BAT -D browser=Firefox -D allstepexecution=No -D
video=Yes -D screenshot=Yes'
/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/lib64/python3.6/site-
packages/azure/storage/blob/_shared/encryption.py:15:
CryptographyDeprecationWarning: Python 3.6 is no longer supported by the
Python core team. Therefore, support for it is deprecated in cryptography
and will be removed in a future release.
  from cryptography.hazmat.backends import default_backend
1 feature passed, 0 failed, 23 skipped
1 scenario passed, 0 failed, 19 skipped
13 steps passed, 0 failed, 437 skipped, 0 undefined
Took 2m31.060s
Allure report was successfully generated.
Creating artifact for the build.
Artifact was added to the build.
Finished: SUCCESS
```

```
Started by user Bekzat Kelgenbayev
Obtained TestAutomationBDDFramework/jenkins/EndToEndTestRDF
[Pipeline] Start of Pipeline
[Pipeline] node
Running on TestAutomation-sim-onprem in /svc/selenium/jenkins-
agent/workspace/testing/RDF/End_To_End_Testing
+ source /svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin/activate
++ deactivate nondestructive
++ '[' -n '' ']'
++ '[' -n '' ']'
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
++ '[' -n '' ']'
++ unset VIRTUAL_ENV
++ '[' '!' nondestructive = nondestructive ']'
++ VIRTUAL_ENV=/svc/selenium/jenkins-agent/workspace/testing/venv_rdh
++ export VIRTUAL_ENV
++ _OLD_VIRTUAL_PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
++ PATH=/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin
++ export PATH
++ '[' -n '' ']'
++ '[' -z '' ']'
++ _OLD_VIRTUAL_PS1=
++ '[' 'x(venv_rdh) ' '!=' x ']'
++ PS1='(venv_rdh) '
++ export PS1
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
+ cd TestAutomationBDDFramework
+ python3 -m runner --job_dir testing/RDF/End_To_End_Testing --run_allure
true '--behave_options=-t Regression -t CancelInstrument -D
environment=June2022-BAT -D browser=Firefox -D allstepexecution=No -D
video=Yes -D screenshot=Yes'
/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/lib64/python3.6/site-
packages/azure/storage/blob/_shared/encryption.py:15:
CryptographyDeprecationWarning: Python 3.6 is no longer supported by the
Python core team. Therefore, support for it is deprecated in cryptography
and will be removed in a future release.
  from cryptography.hazmat.backends import default_backend
1 feature passed, 0 failed, 23 skipped
1 scenario passed, 0 failed, 19 skipped
19 steps passed, 0 failed, 431 skipped, 0 undefined
Took 2m26.079s
Allure report was successfully generated.
Creating artifact for the build.
Artifact was added to the build.
Finished: SUCCESS
```

```
Started by user Bekzat Kelgenbayev
Obtained TestAutomationBDDFramework/jenkins/EndToEndTestRDF
[Pipeline] Start of Pipeline
[Pipeline] node
Running on TestAutomation-sim-onprem in /svc/selenium/jenkins-
agent/workspace/testing/RDF/End_To_End_Testing
```

```
+ source /svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin/activate
++ deactivate nondestructive
++ '[' -n '' ']'
++ '[' -n '' ']'
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
++ '[' -n '' ']'
++ unset VIRTUAL_ENV
++ '[' '!' nondestructive = nondestructive ']'
++ VIRTUAL_ENV=/svc/selenium/jenkins-agent/workspace/testing/venv_rdh
++ export VIRTUAL_ENV
++ _OLD_VIRTUAL_PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
++ PATH=/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin
++ export PATH
++ '[' -n '' ']'
++ '[' -z '' ']'
++ _OLD_VIRTUAL_PS1=
++ '[' 'x(venv_rdh) ' '!=' x ']'
++ PS1='(venv_rdh) '
++ export PS1
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
+ cd TestAutomationBDDFramework
+ python3 -m runner --job_dir testing/RDF/End_To_End_Testing --run_allure
true '--behave_options=-t Regression -t ReactivateInstrument -D
environment=June2022-BAT -D browser=Firefox -D allstepexecution=No -D
video=Yes -D screenshot=Yes'
/svc/selenium/jenkins-
agent/workspace/testing/venv_rdh/lib64/python3.6/site-
packages/azure/storage/blob/_shared/encryption.py:15:
CryptographyDeprecationWarning: Python 3.6 is no longer supported by the
Python core team. Therefore, support for it is deprecated in cryptography
and will be removed in a future release.
  from cryptography.hazmat.backends import default_backend
1 feature passed, 0 failed, 23 skipped
1 scenario passed, 0 failed, 19 skipped
19 steps passed, 0 failed, 431 skipped, 0 undefined
Took 2m45.356s
Allure report was successfully generated.
Creating artifact for the build.
Artifact was added to the build.
Finished: SUCCESS
```