Alliance for Qualification

Released Version 3.0

January 2025

A4Q Selenium 4 Tester Foundation Syllabus

Version 3.0

A4Q
Selenium 4
Tester Foundation

Copyright Notice

Revision History

Version	Date	Remark
1.6	29.10.2022	Initial Merged layout
1.7	15.11.2022	Internally reviewed document
1.8	15.01.2023	TP reviewed document
1.9	26.03.2023	Beta version document
2.0	12.04.2023	Release version
2.1	20.06.2024	Internally reviewed re-written document
2.2	07.07.2024	TP reviewed document
2.3	28.08.2024	Beta version document
3.0	23.01.2025	Release version

Table of Contents

Release

This document was formally released by A4Q GmbH on 30.01.2025.

0 Introduction

0.1 Purpose of this syllabus

This syllabus presents the business outcomes, learning objectives, and concepts underlying the Selenium 4 Tester Foundation training and certification.

0.2 Examinable objectives and cognitive levels of knowledge

Learning objectives support the business outcomes and are used to create the certified A4Q Selenium 4 Tester Foundation exams. A candidate may be asked to recall, show understanding, apply knowledge, and analyze a scenario based on the content from any of the five chapters in the A4Q Selenium 4 Foundation syllabus.

The knowledge levels of the specific learning objectives are shown at the beginning of each chapter, and classified as follows:

- K1: remember
- K2: understand
- K3: apply
- K4: analyze

0.3 The A4Q Selenium 4 Tester Foundation exam

The A4Q Selenium 4 Tester Foundation exam will be based on this syllabus. Answers to exam questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable, except for the Introduction and Appendices. Standards, books, and ISTQB® syllabi may be included as references, but their content is not examinable, beyond what is summarized in this syllabus itself from such standards, books, and ISTQB® syllabi.

The exam shall be comprised of 40 multiple-choice questions. Each correct answer has a value of one point. A score of at least 65% (that is, 26 or more questions answered correctly) is required to pass the exam. The time allowed to take the exam is 60 minutes. If the candidate's native language is not the examination language, the candidate may be allowed an extra 25% (15 minutes) time.

0.4 Accreditation

Training providers wishing to offer training for the A4Q Selenium 4 Tester Foundation should be accredited by A4Q and use the official A4Q Selenium Tester Foundation training materials.

0.5 Level of Detail

This syllabus consists of:

- General instructional objectives describing the intention of the certification.
- A list of terms that students must be able to recall.
- Learning objectives for each knowledge area, describing the cognitive learning outcome to be achieved.
- A description of the key concepts, including references to sources such as accepted literature or standards.

The syllabus content is not a description of the entire knowledge area of Selenium automated testing; it reflects the level of detail to be covered in this foundation level training courses and certification. This syllabus does not contain any specific learning objectives related to any particular software development lifecycle or method.

0.6 How this syllabus is organized

There are five chapters with examinable content. The top-level heading for each chapter specifies the time for the chapter; timing is not provided below chapter level. For the A4Q Selenium 4 Tester Foundation training course, the syllabus requires a minimum of 20 hours of instruction, distributed across five chapters and the practical exercises as follows:

Chapter 1: Introduction to Test Automation – 120 minutes
Chapter 2: Automation Web Technologies – 180 minutes
Chapter 3: Selenium WebDriver – 240 minutes
Chapter 4: Maintainability of TAS and Test Scripts – 150 minutes
Chapter 5: Maximizing ROI on Test Automation – 150 minutes
Practical Exercises: (in Python or Java) - 360 minutes

0.7 Business outcomes (BOs)

Candidates who hold this certification should:

1) Have an understanding of Selenium automation tools
2) Have an understanding of the test automation principles
3) Use the test automation principles to build test automation solutions
4) Be able to evaluate and implement test automation tools
5) Be able to write, execute, troubleshoot, and maintain Selenium WebDriver Script

0.8 Acronyms

Acronym	*Full Form*
AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command Line Interface
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DDP	Defect Detection Percentage
DOM	Document Object Model
EMTE	Equivalent Manual Test Effort
GenAI	Generative Artificial Intelligence
gTAA	Generic Test Automation Architecture
GUI /UI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MBT	Model Based Testing
ML	Machine Learning
MTBT	Mean Time Between Failures
MTTD	Mean Time to Detect
OS	Operating System
POM	Project Object Model
REST	Representational State Transfer
ROI	Return on Investment
SDLC	Software Development Life Cycle
SOAP	Simple Object Access Protocol
SUT	System Under Test
TAE	Test Automation Engineer
TAF	Test Automation Framework
TAS	Test Automation Solution
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

1 Introduction to Test Automation – 120 minutes

Learning Objectives

STF1-1 (K2) Explain the objectives, advantages, disadvantages and limitations of test automation

STF1-2 (K2) Understand the relation between manual and automated tests

STF1-3 (K1) Identify technical success factors of a test automation project

STF1-4 (K2) Explain the place of Selenium WebDriver in TAA

STF1-5 (K2) Understand the risks and benefits of using Selenium WebDriver

STF1-6 (K2) Explain the reason and purpose for metric collection in automation

STF1-7 (K2) Understand and compare objectives of using the Selenium toolset (WebDriver, Selenium IDE, Selenium Grid)

1.1 Test automation in a nutshell

Test automation involves the use of machines to run test scripts on the SUT without the need for continuous human monitoring or intervention. In its simplest terms, it involves using software to facilitate various testing activities, such as test management, test design, test execution, result validation, and reporting outcomes. It is a common misconception that test automation happens only on the graphical user interface level. Test automation can be done at any level of software architecture such as GUI, API, CLI, etc.

While test analysis, design, and implementation are often manual processes, there are opportunities to automate these activities using Model-Based Testing (MBT) or Generative Artificial Intelligence (GenAI). Autonomous testing is an emerging technology that leverages AI/ML and MBT to interpret user stories, extract test conditions, test scripts, create test data, and execute tests without human intervention. However, it may take some time for this technology to fully mature and achieve widespread adoption.

Test automation requires a supporting ecosystem that consists of various components, including:

- The test environment
- The tools employed
- The test automation library
- The test automation engine
- The test scripts
- The test harnesses
- The test reporting mechanism to generate test reports

The main objectives of test automation include:

1. **Increased Testing Efficiency**: Automating repetitive and time-consuming test cases helps reduce the overall time required for testing, especially during regression cycles.
2. **Improved Accuracy**: Automation minimizes the risk of human errors that may occur during manual testing, ensuring more reliable test results.
3. **Enhanced Test Coverage**: Automated tests can cover a larger number of test cases, including complex scenarios that might be difficult or impractical to test manually.
4. **Cost Reduction Over Time**: Although initial setup costs for automation can be high, the long-term savings in labor and faster test cycles reduce the overall cost.
5. **Reusability**: Test scripts can be reused across different versions of the application, improving scalability and adaptability to changes.
6. **Faster Feedback**: Automation provides immediate feedback on the quality of the application after code changes, allowing developers to fix issues earlier in the development cycle.
7. **Support for Continuous Integration/Delivery (CI/CD)**: Automation integrates seamlessly into CI/CD pipelines, enabling faster and more reliable deployments.

8. **Consistency in Execution**: Automated tests run the same steps consistently, which eliminates variability in results due to differences in human execution.

The main benefits and advantages of test automation include:

- Running automated tests may be more efficient than running them manually.
- Ability to perform tests that are difficult or impossible to do manually, such as reliability or performance tests.
- Reduction in the time required for test execution, enabling test engineers to run more tests per build.
- Increased frequency of test execution.
- Increased test coverage through different test runs with different test configurations (e.g. SUT language or cross-browser tests)
- Freeing up manual test engineers to execute more interesting and complex manual tests (e.g., exploratory tests).
- Reduction in errors induced by human distraction, especially when repeating regression tests.
- Execution of tests earlier in the process (e.g., integration in the CI/CD pipeline of automatic unit, component, and integration tests), leading to quicker feedback on system quality and early defect removal.
- Ability to run tests outside of normal business hours.
- Increased confidence in the quality of the build.
- Ability to push more testing earlier in the SDLC to detect and eliminate defects from the code sooner, thus reducing MTTD (Mean Time to Detect).

Test automation needs a solid strategy without which the following challenges and disadvantages could be encountered:

- Increase in costs (high setup and recurring maintenance costs make it difficult to break even)
- Delays, costs, and mistakes associated with test engineers as they learn new technologies.
- Increasing complexity provides challenges in transparency and maintainability.
- Unacceptable growth in the size of the automated tests, potentially exceeding the size of the SUT.
- Extensive software development skills are needed to sustain the project.
- Considerable maintenance of tools, environments, and test assets is required.
- Accumulation of technical debt, especially when extra programming is added to improve the intelligence of the automated tests.
- Test automation requires all the processes and disciplines of software development.
- Risk of test engineers losing sight of risk management for the project by concentrating solely on test automation.
- Risk tests wear out over time, especially when test automation is used, as exactly the same test is executed each time. This is also commonly known as the Pesticide Paradox

- Occurrence of false positives and false negative with test automation, where failures are due to defects in the test automation framework itself (false positive) or lack of verification in the automated tests (false negative) whereby defects are missed.
- Without clever programming in the automated tests, tools are unable to figure out what happened and how to recover from unexpected behavior.

Several factors can affect a test automation project, some of which can be alleviated through diligent programming and adherence to good engineering principles, while others cannot be changed. These limitations include technical and managerial aspects such as:

- The required skills and knowledge need to be available.
- Proper configuration management (bidirectional) needs to be in place and maintained.
- Unrealistic expectations by management often drive automation in the wrong direction.
- Short-term thinking can harm a test automation project, and long-term planning is essential for success.
- Organizational maturity is required to succeed; test automation based on poor testing processes only delivers bad testing faster (and sometimes even slower).
- Some test engineers are happy with manual testing and do not want to automate.
- Automated test oracles may differ from manual test oracles, adding another level of complexity.
- Not all tests can or should be automated.
- Manual testing will still be required (exploratory testing, certain fault attacks, etc.).
- Human beings find most defects; test automation can only find what it is programmed to find.
- A false sense of security is created due to large numbers of automated tests running without finding many defects.
- Technical problems may be introduced in projects when using cutting-edge tools or techniques.
- Cooperation between test engineers and the development team, which can create organizational issues.

Test automation can and often does succeed and it is because of attention to detail, good engineering practices, and hard work over the long term.

1.2 Test automation and manual testing

Manual testing has traditionally been the most reliable method for testing. Automation tools offer many helpful features to simplify testing activities, but their effectiveness depends greatly on how well the automated test script is implemented.

To illustrate this further, consider the basic manual test case below:

Actions	Test Data	Expected Result
Navigate to website	https://www.google.com/	Website is well loaded.
Login on the website	Username: user@gmail.com Password: secret	User credentials are accepted, and the homepage is displayed.
Click on logout button	-	The user is logged out and the user is redirected to the login page.

The test case above can be automated using a record/playback tool, but there are several potential reasons why the script could fail such as:

1) Timing issues: The website may take slightly longer to load, causing the automation tool to attempt to login while the page is not fully loaded
2) GUI changes: The properties of the elements on the page may change, meaning the automation tool can no longer locate the elements to interact with.
3) Browser properties: The updated version of the browser may not be compatible with the automation tool.

An experienced manual tester brings context and reasonableness to the test execution. In the example above:

Context: The tester knows where to enter the username, the password, and what to click on to login in the platform. This information needs to be explicitly defined in the automation tool to describe the context.

Reasonableness: The tester understands the acceptable period to wait for the website to load. For instance, if the automation tool is programmed to wait for 3s for the page to load, but the page takes 3.1s, the test case may fail.

The context and reasonableness of the test must be clearly defined for automated execution. If these criteria are not established, the test may fail and produce a false-positive result. A false positive occurs when a defect is reported even though no actual defect exists in the test object.

As a result, it is often better for the users to handle unexpected test outcomes rather than relying solely on the automation tool. The reasonableness and context can be programmed into the automation tool to maximize the value created by the tool, but this eventually adds to the maintenance effort required on the tool. There is therefore a trade-off to consider in this situation.

It is a common misunderstanding that all manual test cases need to be automated. Maintaining automated test scripts usually requires more effort than manual test cases. Therefore, it's important to carefully select which test cases to automate. The most important criteria for selecting test cases to automate includes:

1) Frequency of execution: The more frequently a test case is executed, the greater the value gained from automating it. This is why regression test cases are typically the best candidates for automation.

2) The stability and consistency of the SUT's UI, features and test environment: If parts of the SUT are not subjected to regular and intensive changes, then automating tests, related to these stable parts of the SUT, creates more value as the maintenance effort of those tests will be reduced.

Considering the above, it can be concluded that some manual tests will remain because there is no positive return on investment (ROI) for automating them. Additionally, there are some manual tests that cannot be automated because the thought process of the manual tester is essential to the success of the test. Exploratory testing, fault attacks, and other types of manual testing are still required for the success of a testing effort.

1.3 Success Factors

The ability to automate often depends on the testability of the SUT. In some cases, the SUT's interfaces may not be suitable for testing. Obtaining special or private interfaces (test hooks) from the developers of the system can make the difference between succeeding or failing at test automation.

Several different levels of interface can be used to interact with the SUT:

1. The GUI level (often the most brittle and failure-prone level)
2. The API level (using application programming interfaces that the developers make available for public or protected use)
3. Private hooks (APIs that are specifically for testing)
4. The protocol level (HTTP, TCP, etc.)
5. The service level (SOAP, REST, etc.)

It is important to note that Selenium works at the GUI level. This syllabus contains tips and techniques designed to reduce brittleness and enhance usability while testing at this level.

The success of test automation heavily depends on the nature of the software project. By following best practices in the engineering processes, the benefits of test automation can be maximized while its drawbacks can be minimized. It is important to note that success in automation does not happen by accident.

The skills of test engineers (knowledge, experience and practices), the proper implementation of tools, and the supporting metrics of the tools (efficiency, effectiveness, and return on investment) are some important factors that contribute to the success of test automation projects in an organization.

The following are key success factors for test automation:

1. Management needs to have a clear understanding of what is feasible and what is not. Unrealistic expectations from management are one of the primary reasons why software test automation projects fail.

2. The development team for the SUT(s) should understand automation and be willing to collaborate with the testers when necessary.
3. A mature test process and sound documentation about test automation engineering best practices should be in place.
4. The SUT(s) should be designed for testability.
5. A business case for the short, medium, and long-term goals should be developed.
6. The right tools to work in the environment and with the SUT(s) should be selected.
7. The right training, including both test and development disciplines, should be provided. This should include both theoretical and practical knowledge and knowhow.
8. The architecture and framework supporting individual test scripts i.e. the TAS should be well-designed and thoroughly documented.
9. A well-documented test automation strategy that is funded and supported by management is drafted.
10. A formal and well-documented plan for maintenance of the test automation solution is set in place.
11. A practice of automating at the right interface level for the context of tests is enforced to maximize the efficiency, effectiveness and ROI of the test automation solution.

1.4 Risks and Benefits of Selenium WebDriver

Selenium WebDriver is a collection of language-specific bindings that provide a programming interface for developing advanced and robust Selenium scripts used to drive the web browser.
Selenium WebDriver supports several programming languages. Some popular ones are listed below:

- Java
- Python
- C#
- Ruby
- JavaScript
- Perl
- PHP
- R
- Dart

Many of these languages also have open-source testing frameworks available. Web browsers supported by Selenium include:

1. Google Chrome (Chromium based)
2. Microsoft Edge (Chromium based)
3. Mozilla Firefox
4. Opera (Chromium based)
5. Apple Safari

The Selenium WebDriver is a W3C recommendation which implies that Selenium WebDriver has undergone a rigorous review process and has been accepted as a formal standard for how web browsers should be controlled programmatically. The recommendation ensures that browser vendors adhere to this standard, leading to more consistent behavior across different browsers when using Selenium WebDriver for automated testing. W3C recommendation status also suggests that:

1. Selenium WebDriver is designed as a simple and more concise programming interface.
2. Selenium WebDriver is a compact object-oriented API.
3. Selenium WebDriver drives the browser effectively.

Selenium WebDriver works by using its APIs (Application Programming Interfaces) to make direct calls to a browser using each different browser's native support for test automation. Each supported browser works slightly differently. As with any tool, there are both benefits and risks to using Selenium WebDriver. Below are the main benefits of the Selenium WebDriver:

1. Test execution can be consistent and repeatable.
2. Well suited for regression tests.
3. Because it tests at the UI level, it may catch defects missed by testing at the API level.
4. Lower up-front investment because it is open source.
5. Works with different browsers so compatibility testing is possible.
6. Supports different programming languages so it can be used by people from different programming backgrounds.
7. Because it requires a deeper understanding of the code, it is useful for Agile teams as then the developers and test automation engineers share common understanding of SUT features and their implementations.
8. Selenium WebDriver allows automation of complex user scenario including complex UI interactions like drag and drop.
9. Selenium WebDriver integrates well with testing frameworks like TestNG, Junit, PyTest, etc.
10. Selenium WebDriver is supported by a large community thus having extensive resources and libraries.
11. Selenium WebDriver supports cross-browser testing on Google Chrome, Mozilla Firefox, Edge, etc.
12. Tests can be scaled through parallel test execution on various machines and browsers using Selenium Grid.
13. Selenium WebDriver offers customization and flexibility as testers can write custom logic for specific testing needs.
14. Headless testing is possible therefore making test execution faster.
15. Robust API of Selenium WebDriver allows precise control over the test automation.
16. Selenium WebDriver integrates with tools like Appium making mobile testing possible.
17. Selenium WebDriver integrates seamlessly with CI/CD tools like Jenkins, Azure DevOps, GitLab CI.
18. Selenium WebDriver provides detailed stack traces and error message which aids in debugging. It can further integrate with third party libraries to provide enhance reporting.

The following are risks and limitations that come with the use of Selenium WebDriver.

1. Organizations often get so wrapped up in GUI testing that they forget that the testing pyramid suggests more unit/component testing is done.
2. When testing for a CI (continuous integration) workflow, this automation may make the build take much longer to complete than desirable.
3. Changes to the UI cause more damage to browser-level tests than they do to unit or API level tests and testers get into long cycles of automation test maintenance.
4. Manual testers are more effective at finding bugs than test automation is.
5. Tests that are difficult to automate might get skipped.
6. Automation needs to be run often to return a positive ROI (return on investment). If the web application is fairly stable and consistent, then test automation may not pay for itself.
7. There can be browser compatibility issues as different browsers behave differently.
8. There is limited built-in reporting in Selenium WebDriver and relies a lot on third party libraries for such functionalities.
9. Selenium WebDriver needs programming skills to write, maintain and debug test scripts.
10. Selenium WebDriver struggles to handle natively non-html elements such as file uploads, downloads and some OS level popups.

1.5 Selenium WebDriver in Test Automation Architecture

The TAA (Test Automation Architecture) is a set of layers, services, and interfaces of a TAS (Test Automation Solution). TAS is the implementation of a test automation architecture for a test automation assignment. The TAA consists of four layers:

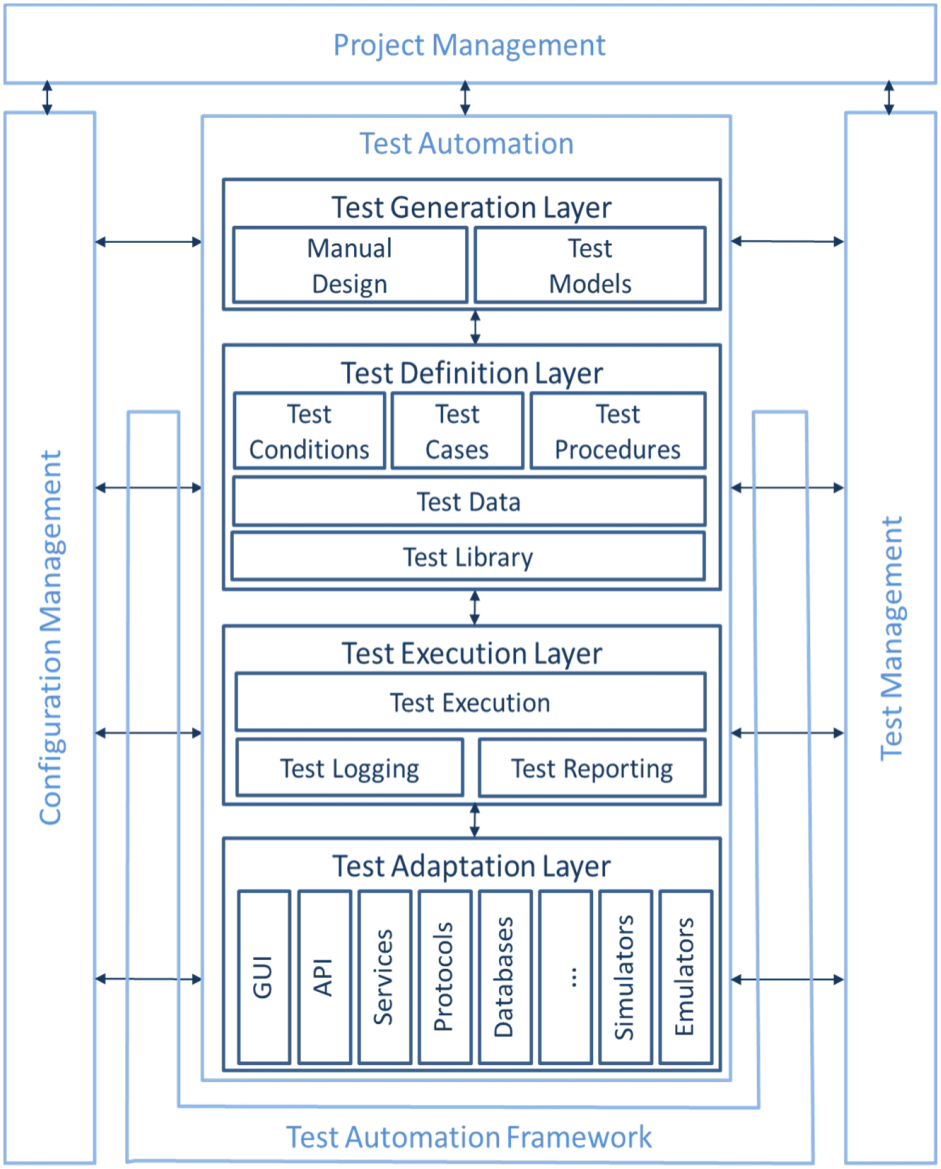
1. The test generation layer supports the manual or automated design of test cases
2. The test definition layer supports the definition and implementation of test cases and /or suites
3. The test execution layer supports both the execution of automated tests and the logging/reporting of results
4. The test adaptation layer provides the necessary objects and code to interface with the SUT (system under test) at various levels.

Selenium WebDriver fits into the test adaptation layer, providing a programmatic way to access the SUT through the browser interface. The test adaptation layer facilitates a separation of the SUT (and its interfaces) from the tests and test suites we want to run against the SUT. Ideally, this separation allows the test cases to be loosely coupled and disconnected from the system that is being tested. When the system under test changes, the test cases themselves may not need to change, assuming that the same functionality is being provided in a slightly different way. When the SUT changes, the test adaptation layer, in this case, WebDriver, allows modification of the automated test, allowing it to run the actual concrete test case against the modified interface of the SUT.

The automated script using WebDriver effectively uses the API to communicate between the test and the SUT as follows:

1. The test script calls for a task to be executed
2. The script calls some API functions in WebDriver
3. The API connects to the appropriate object in the SUT
4. The SUT responds as to the API call made.
5. Success or failure of the action is communicated back to the script by Selenium WebDriver
6. If successful, the next step in the test case is called in the script else the current step is marked as failed.

Below is the block diagram of gTAA (Generic Test Automation Architecture) with all the test layers.



A generic TAA (from ISTQB® Test Automation Engineer Syllabus)

1.6 Purpose for Metrics Collection in Test Automation

Measurement in test automation can often be a challenging task, leading many test automation engineers to overlook it. The Goal-Question-Metric (GQM) framework is highly effective in defining and tracking test automation metrics by aligning them with specific objectives.

In test automation, GQM starts with identifying high-level goals, such as improving test coverage, reducing test execution time, or enhancing test reliability. From these goals, targeted questions are formulated to probe key areas, such as "How frequently are automated tests failing?" or "Is test automation reducing manual testing effort?" Finally, metrics are established to provide quantitative answers to these questions, including metrics like test pass/fail rates, time saved, defect leakage, and automated test coverage. By linking test automation metrics directly to organizational goals, the GQM approach helps teams focus on meaningful improvements and demonstrate the value of automation efforts in a structured, data-driven manner.

Test automation requires significant human effort and investment in tools, making it an expensive undertaking. It may not show an immediate positive return on investment, especially if implemented late. The real benefits of test automation are usually realized in the long term. This is why management support is needed, and quantitative metrics are used to track progress. To strengthen management support for test automation, a strong business case based on meaningful collected metrics is essential. These metrics should provide an estimation of the potential value of test automation.

Key areas for collecting metrics to estimate the value of test automation projects include:

1. Performance: Allows benchmarking the time saved compared to manual test execution
2. Reliability: Shows the ratio of false positive fails to defects
3. Maintainability: Indicates the recurrent cost and effort for test and TAS maintenance
4. Usability: Indicates the ease of use of TAS and how likely to include manual testers

A valuable metric used in test automation is EMTE (Equivalent Manual Test Effort), as the name suggests is the equivalent effort needed for running tests manually. Let us consider a pilot test automation project that runs an automated test in 15 minutes. The same test case takes 30 minutes to execute manually. Therefore 30 minutes is the EMTE of the execution of the test case. This allows us to estimate that if an automated test takes 3 hours to execute, the EMTE would be around 6 hours. Comparing the EMTE of test case execution to the time needed to automate it makes sense. When the effort required to automate the test outweighs the effort of manual execution after multiple runs, the organization realizes a return on investment (ROI). The EMTE metric is an example of a performance metric.

It is important to note that when estimating test automation effort, the following factors need to be considered:

1. Effort related to evaluating test cases (check if automatable and if automation will create value)
2. Effort related to test script design (how to automate the test case)

3. Effort related to automating the test case
4. Effort related to automate test script debugging
5. Effort related to automated test case maintenance

Important questions that need to be answered by the test automation engineers (TAEs) to define the value of automating a test are:

1. How much will it cost to automate the test case?
2. What is the frequency of updates on the GUI of the SUT?
3. How many times in a defined period can the test fail due to false positive result?
4. What verification points should be added to avoid false negative results?
5. What is the frequency of execution of the test?
6. By how much are risks of failure of the software reduced?
7. What will be the coverage levels of the automated tests?

The TAEs also need to consider the cost of acquiring automation tools and the skillset to use the tools effectively. This constitutes the fixed cost of the test automation project. It is easier to achieve a positive ROI when a large number of frequently executed tests is automated.

Based on the above, some metrics are important to be considered in an automation project. For example:

1. EMTE metrics of tests
2. Fixed costs to get the test automation solution up and running
3. Regression test effort that has been saved by the automation
4. The effort invested by the test automation team supporting the automation project
5. Time to detect defects (within areas covered by automation tests) after code change.
6. The MTBF metrics to understand how period automation executions increase reliability of the SUT through early and regular defect detection.
7. Automated test coverage
    - At the unit test level, via statement/decision coverage
    - At the integration test level, interface or data flow coverage
    - At the system test level, requirement, feature, or identified risk coverage
    - User stories covered (in Agile)
    - Use cases covered
8. Covered configurations tested by the automation solution
9. Number of successful runs between failures
10. Patterns of automation failures (looking for commonality of problems by tracking root causes of failures)
11. Number of test automation failures (false positive results) found as compared to SUT failures found by the automation solution

Metrics collected for an automation project can differ from one project to another. However, the main focus should be on gathering meaningful metrics that provide value to higher management and support the project's goals. Some metrics can be benchmarked against current manual testing processes. For instance, metrics such as the frequency of regression test executions and the number of defects missed in the test environment can be obtained from the manual testing team. Thus, it's advisable to initiate a test automation project when the manual testing process is sufficiently mature.
In a nutshell, an automation project can be very successful if implemented correctly although quantifying the success of the project can be quite difficult.

1.7 The Selenium Toolset

The Selenium ecosystem evolves around 3 main tools:

1. Selenium IDE
2. Selenium WebDriver
3. Selenium Grid

Selenium IDE is a browser extension or plugin that integrates with popular browsers such as Google Chrome and Mozilla Firefox to allow the user to easily record manual steps done on the browser and play the recorded steps multiple times repeatedly.
The new Selenium 4 IDE provides some notable features including:

1. Improved GUI for intuitive user experience.
2. The new IDE comes bundled with a SIDE tool (Selenium IDE) runner. It allows software testers to run '.side' projects on a node.js platform. This SIDE runner also enables individual test engineers to run cross-browser tests on local or Cloud Selenium Grid.
3. An improved control flow mechanism that enables testers to write better "while" and "if" conditions.
4. The new IDE comes with an enhanced element locator strategy (like a backup strategy) which helps locate an element in case the web element couldn't be located. It will result in the creation of stable test cases.
5. The code for test cases recorded using Selenium IDE can be exported in the desired language binding such as Java, C#, Python, .NET, and JavaScript.

Selenium IDE is suitable for short-term tests or automation needs. The simplicity of the tool compensates for its lack of extended flexibility. However, the latest version of Selenium IDE is more resilient for tests. This is because when an interaction with a web object occurs, multiple locators for the object are recorded and during playback, each locator is attempted. There is therefore better resilience in the scripted tests. There is also better re-use of existing test cases in the new Selenium IDE. This means that existing test cases could be called in other test cases without duplication. The re-use functionality therefore reduces the test maintenance effort as the called test cases are shared and not duplicated.

The new Selenium IDE has constructs that allow better control and execution flow. Controls like the if statement and looping structure come in very handy when we need to implement real-life tests that are based on some logical flow. The extensibility of the new Selenium IDE is appealing as it allows custom commands to be written and third-party plugins to be integrated as well.

Nevertheless, if higher control flow, flexibility, and extensibility are needed, the Selenium IDE may not be the best tool. Selenium WebDriver is better when it comes to these needs.
Selenium WebDriver interacts with the browser in the same way that a real user would. WebDriver is an end-to-end test compilation tool for web applications. The language bindings and the actual implementation of the unique browser controlling code are collectively referred to as Selenium WebDriver. The WebDriver performs exactly what a user would anticipate from a browser: it automates browser control so that a user can repeat the automated actions. Although it appears to be a straightforward challenge to overcome, several steps must be taken for it to function. The Selenium WebDriver is the interface in the TAS that allows interaction with the browser.

The Selenium WebDriver is a major building block for TAS. It runs the browser significantly efficiently whether it is operated by a local user or a remote user. As a result, the functional test coverage constraints of the previous version of Selenium, such as those related to file uploads, downloads, pop-ups, and dialogue barriers, are removed. The language bindings and implementations of the separate browser controlling code have been merged into Selenium WebDriver. The Selenium WebDriver is entirely an object-oriented API. The implementation of WebDriver normally differs for each browser. Numerous language bindings are supported by Selenium WebDriver and work with modern web browsers.

From WebDriver architecture, it is known that WebDriver directly calls the browser when tests are performed in a native machine by using the browser's built-in JavaScript support for the automation. Thus, WebDriver does not have any proxy server between the client and the browser.
The WebDriver API serves as a powerful tool for accelerating test execution, and it supports nearly all recent browser versions available in the market.

The use of Selenium WebDriver for test automation brings about some advantages for organizations. These include:

1. Selenium WebDriver is a collection of crude API interfaces that can be adapted to the organization's needs.
2. Additional adapters, such as those for database connections and web service connections, can be incorporated to enhance the effectiveness of the tests.
3. The TAS based on Selenium WebDriver, if well-designed, can be highly flexible in the long term.
4. The availability of an off-the-shelf TAS based on Selenium WebDriver gives a quicker implementation of automation testing in organizations.

One drawback associated with using Selenium WebDriver is that sound technical knowledge and skill are needed to make effective use of Selenium WebDriver.

Selenium Grid empowers the Selenium WebDriver to run tests on multiple machines with varying configurations concurrently. This has numerous advantages. The first one is that the time of test execution is greatly reduced. Given that a long-running test or a set of test suites can be executed concurrently on different machines, it means that the time taken for tests to complete will be significantly reduced. Nevertheless, the tests need to be carefully scripted to ensure that the test suites are independent of each other. The next major advantage is that the automated test coverage levels with respect to the different test configurations (devices, browsers & versions, and the OS) can easily be increased. Given that Selenium WebDriver supports test execution on a wide range of browsers, versions, and operating systems, Selenium Grid exploits this capability to allow concurrency in test execution. At the heart of the grid, the hub machine controls the other node machines connected to it. The node machines execute the test scripts instructed by the hub independent of the other nodes.

Earlier versions of Selenium Grid were complex to implement and rigid in terms of scaling. However, this new version comes with Docker support. This will allow TAEs (or developers) to spin up the containers rather than setting up multiple machines with high-end configurations.
Managing Selenium Grid is now smoother and easier as there is no longer any need to set up and start hubs and nodes separately.

The Grid can now be deployed in 3 modes:

- **Standalone Mode:** A single machine acts as both the hub and the node, making it simple to set up. Best for local or small-scale testing, but it lacks scalability.
- **Hub and Node Mode:** A central hub distributes tests to multiple nodes, allowing distributed execution across different machines and environments. It's scalable and flexible but requires more setup and maintenance.
- **Distributed Mode:** Introduced in Selenium Grid 4, it decouples components (e.g., router, distributor, nodes) for high scalability and fault tolerance. Ideal for enterprise-level or large-scale testing but involves complex configuration.

Unlike earlier versions, the Grid will now support IPv6 addresses, and one can communicate with the Grid using the HTTPS protocol. In Grid 4, the configuration files used for spinning up the grid instances can be written in TOML (Tom's Obvious, Minimal Language) which will make it easier for humans to understand.
The Grid in Selenium 4 also comes with an enhanced user-friendly GUI. Overall, the revamped Selenium Grid will enhance the DevOps process as it provides compatibility with tools like Azure, AWS, and more.

2 Automation Web Technologies – 180 minutes

Learning Objectives

STF2-1 (K3) Analyze HTML and XML documents

STF2-2 (K3) Apply XPath to search XML documents

STF2-3 (K3) Apply CSS selectors to find elements in HTML documents

2.1 Understanding HTML

HTML (HyperText Markup Language) played a pivotal role in shaping the World Wide Web. An HTML document, essentially a plain text file, contains elements that carry specific contextual meanings when parsed. These elements collaborate to instruct a browser on how to display different parts of the document, effectively outlining the structural semantics of a web page.

One of HTML's key advantages lies in its universal applicability. Properly authored HTML can be rendered correctly by any browser on any computer system. However, the visual appearance of the page may vary depending on factors such as the device (e.g., PC versus smartphone), monitor, internet connection speed, and browser.

HTML elements are introduced and often surrounded by tags which are defined by angle brackets as seen below:

```
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Sample HTML Page</title>
    </head>
    <body>
        <header>
            <h1>Welcome to My Website</h1>
        </header>
        <main>
            <p>This is a sample HTML page.</p>
        </main>
        <footer>
            <p>&copy; 2024 MyWebsite.com.</p>
        </footer>
    </body>
</html>
```

Certain tags directly add content to the rendered page, such as , which inserts an image. Conversely, other tags encapsulate and define semantic information regarding how the element should appear, as exemplified by <h1>...</h1> for a heading element.

In 2014, the governing body overseeing HTML introduced the current version, HTML5. HTML offers some flexibility, permitting variations in tag usage, like omitting closing tags. This flexibility has led to erratic page rendering in some browsers. In contrast, XML imposes stricter rules, mandating "well-formed" pages with

balanced opening and closing tags. When written in a well-formed manner—where all opening tags have corresponding closing tags, HTML functions become a subset of XML.
Efficient automation with Selenium depends on understanding HTML tags. To automate GUIs effectively, the test automation engineer must identify each unique control within the manipulated screen. Navigating through the HTML page enables the test automation engineer to distinctly pinpoint the controls presented on the browser-rendered page. To locate these controls, one must grasp the layout and logic of the HTML page, accomplished by comprehending and parsing through its tags. Typically, HTML elements comprise both start and end tags. The end tag mirrors the start tag but is prefixed with a slash, as demonstrated below:

```
<p>Paragraph text</p>
```

Some elements can be closed within the open tag; for example, the empty line break element as follows:

A less strict implementation of the line break, which might cause problems for some browsers would be:

The following are tags that every Selenium test engineer should understand.

Tag	Used for
<html> … </html>	Represents root of HTML document
<head> … </head>	Definition and meta data for document
<body> … </body>	Defines main content for the document
** … **	Defines a paragraph
** **	Inserts a single line break
** … **	Defines a section in the document
<!-- … -->	Defines a comment (may be multi-line)

Heading tags define different levels of headers. The actual format of the text (size, boldness, font) can be specified in CSS stylesheets.

Links and images play a crucial role in the effective design of web pages, and they can be effortlessly generated using HTML.

```
<a href="URL">link text</a>
```

This sequence of symbols begins with an opening <a ...> anchor tag and concludes with a closing tag. Together, they define a hyperlink that users can click on. The href="URL" attribute indicates the destination of the link. The text enclosed between the tags represents the clickable link text that directs users to the specified URL when clicked.

```
<img src="a4q.jpg" alt="A4Q Logo" />
```

The fundamental tag, <img.../>, specifies an image to be positioned at the designated spot within the document. The src="a4q.jpg" attribute denotes the web address of the image file to be displayed. The alt="A4Q Logo" attribute provides alternative text that will appear in case the image cannot be located or rendered.

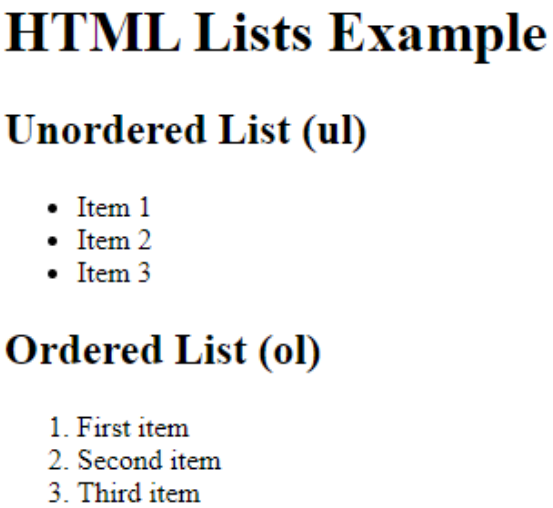
Tag	Used for
** … **	Defines an unordered (bulleted) list
** … **	Defines an ordered (numbered) list
** … **	Defines a list item (for or
<table> … </table>	Defines an HTML table
<tr> … </tr>	Defines a table row
<th> … </th>	Defines the column header for a table
<td> … </td>	Defines a table data cell
<tbody> … </tbody>	Groups body content in an HTML table
<thead> … </thead>	Defines an HTML table header
<tfoot> … </tfoot>	Defines an HTML table footer
<colgroup> … </colgroup>	Groups table columns for formatting

Creating lists in HTML is straightforward, and the code snippet below will display the subsequent list:

```
<html lang="en">
        <head>
         <meta charset="UTF-8">
         <title>HTML Lists Example</title>
        </head>
        <body>
                <h1>HTML Lists Example</h1>
                <h2>Unordered List (ul)</h2>
                        <ul>
                                <li>Item 1</li>
                                <li>Item 2</li>
                                <li>Item 3</li>
                        </ul>

                <h2>Ordered List (ol)</h2>
                <ol>
                         <li>First item</li>
                         <li>Second item</li>
                         <li>Third item</li>
                </ol>
        </body>
</html>
```

This code snippet creates an HTML document with a heading, an unordered list with three items, and an ordered list with three items and is rendered as below:



Creating and rendering tables in HTML is straightforward. Tables are commonly used to display test result data, making them essential for test automation engineers.

The provided code will display the table as follows:

```
<html lang="en">
	<head>
		<meta charset="UTF-8">
		<meta name="viewport" content="width=device-width, initial-scale=1.0">
		<title>Sample Table</title>
		<style>
			table {
				width: 100%;
				border-collapse: collapse;
			}
			th, td {
				border: 1px solid black;
				padding: 8px;
				text-align: left;
			}
			th {
				background-color: #f2f2f2;
			}
		</style>
	</head>
	<body>
		<table>
			<tr>
				<th>Name</th>
				<th>Age</th>
				<th>Country</th>
			</tr>
			<tr>
				<td>John Doe</td>
				<td>30</td>
				<td>USA</td>
			</tr>
			<tr>
				<td>Jane Smith</td>
				<td>25</td>
				<td>Canada</td>
			</tr>
			<tr>
				<td>Alice Johnson</td>
				<td>35</td>
				<td>UK</td>
			</tr>
		</table>
	</body>
</html>
```

This code creates a table with three rows (`<tr>`) and three columns (`<th>`), with each row containing data (`<td>`) for Name, Age, and Country. The table is styled using CSS to have borders around cells and a gray background for header cells.

Name	Age	Country
John Doe	30	USA
Jane Smith	25	Canada
Alice Johnson	35	UK

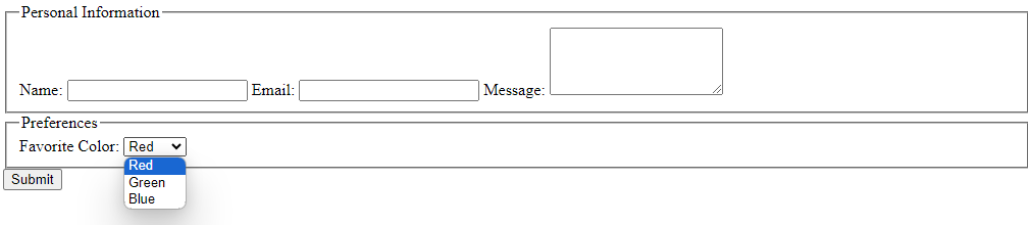
HTML forms and their corresponding controls are utilized to collect input from users. Below are the necessary tags required to display the forms and the controls within them. Selenium WebDriver users frequently need to engage with these forms and controls to automate their testing processes.

Tag	Used for

<form>…</form>	Defines an HTML form for user input
<input>	Defines an input control. Possible types include text, radio, checkbox, submit, etc.
<textarea>…</textarea>	Defines a multiline input control
<button>	Defines a clickable button
<select>…</select>	Defines a drop-down list
<fieldset>…</fieldset>	Defines grouping of related elements into a form.

```
<html lang="en">
        <head>
          <meta charset="UTF-8">
          <meta name="viewport" content="width=device-width, initial-scale=1.0">
          <title>Sample Form</title>
        </head>
        <body>
        <form action="/submit-form" method="post">
          <fieldset>
            <legend>Personal Information</legend>
            <label for="name">Name:</label>
            <input type="text" id="name" name="name" required>
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" required>
            <label for="message">Message:</label>
            <textarea id="message" name="message" rows="4" required></textarea>
          </fieldset>
          <fieldset>
            <legend>Preferences</legend>
            <label for="color">Favorite Color:</label>
            <select id="color" name="color">
              <option value="red">Red</option>
              <option value="green">Green</option>
              <option value="blue">Blue</option>
            </select>
          </fieldset>
          <button type="submit">Submit</button>
        </form>
        </body>
</html>
```

The above code will generate the following control set



2.2 Understanding XML

XML (eXtensible Markup Language) serves as a markup language aimed at defining rules for document formatting in a manner that is both machine-readable and human comprehensible. Its design emphasizes simplicity and usability across the World Wide Web. Unlike HTML, which focuses on presenting data visually, XML allows for the representation of diverse data structures that can be created dynamically. XML was crafted to function as a software and hardware-independent tool for transporting and storing data in a readable format. Unlike HTML tags, XML tags are not predefined; rather, they are defined by the XML document creator according to their chosen standards. The tag format closely resembles that of HTML, providing a familiar structure for users.

For example, below is a set of fields represented in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder>
  <orderID>PO-2024-04-28</orderID>
            <customer>
              <name>Acme Corporation</name>
              <address>
                <street>123 Main Street</street>
                <city>Anytown</city>
                <state>CA</state>
                <zip>98765</zip>
              </address>
            </customer>
  <items>
              <item>
                        <sku>12345</sku>
                        <description>Stapler</description>
                        <quantity>10</quantity>
                        <price>5.99</price>
              </item>
              <item>
                        <sku>ABCDE</sku>
                        <description>Box of Printer Paper</description>
                        <quantity>2</quantity>
                        <price>12.50</price>
              </item>
            </items>
  <total>33.98</total>
</purchaseOrder>
```

In this example:

- customer and items are child elements of the purchaseOrder element.
- name, address, item, etc. are further nested elements providing details about the customer and order items.
- Attributes like orderID and sku can be directly added to elements for specific data points.
- This demonstrates how XML can represent hierarchical data structures effectively.

For each opening tag, there is a closing tag. The total construct (from the opening to closing tags) is called an element. An XML document always forms a tree structure. In the above example, the root element of the tree is purchaseOrder.

In addition to tags, XML supports attributes which supply extra information about the element they are associated with. An attribute consists of a pair of terms separated by an equal sign.
For example:
<person nationality=”Mauritian”>

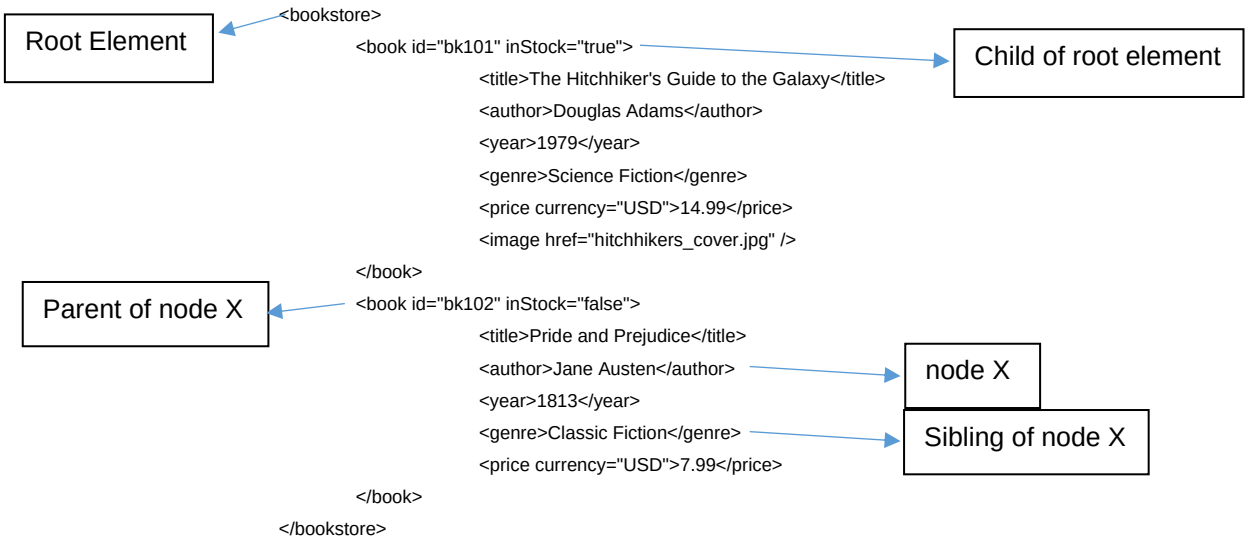
The attribute is contained within the element’s brackets. Rather than using an attribute, the same information can be used as an element.
The below two XMLs are equivalent:

```
<person>
        <nationality>Mauritian</nationality>
        <firstname>Lovelesh</firstname>
        <lastname>Beeharry</lastname>
</person>
```

```
<person nationality=”Mauritian”>
        <firstname>Lovelesh</firstname>
        <lastname>Beeharry</lastname>
</person>
```

Attributes lack the flexibility of elements, as emphasized by the W3C, the organization overseeing the XML standard. They have several limitations: they cannot accommodate multiple values (unlike elements), they cannot represent tree structures (as elements do), and they are not easily extensible for future modifications.

Given the diverse ways computers store data, often incompatible with each other, XML serves as a bridge, facilitating data sharing through plain text format. Unlike the complex handshakes required for communication between computers, XML allows them to communicate seamlessly using text files. By separating data from presentation, XML enables authors to present the same data in various formats according to their preferences.



Since XML always describes a tree structure, we can define specific relationships between elements. These relationships can be defined as follows:

1. The current node is any arbitrary node that we choose. All other relationships derive from the current node.
2. Each node can identify itself as self.
3. A parent node is always one level higher in the hierarchy than the node selected as current. Each element node and attribute has exactly one parent (except for the root element.)
4. A child node is always one level below its parent in the hierarchy.
5. A sibling node is on the same level as the current node, under the same parent.
6. An ancestor node is one in a direct path from the current node up through its parent, grandparent, great-grandparent, etc.
7. A descendent node is one in a direct path down from the current node in the hierarchy (i.e., a child, a child of a child, etc.)

2.3 XPath

For automation to be possible, the test automation engineer must be able to locate objects or elements in an XML document. One of the most powerful ways to do so is to use **XPath**.

XPath stands for XML Path Language. It uses a non-XML syntax to provide a flexible way to navigate and identify nodes in an XML document. Since well-formed HTML is a subset of XML, XPath can also be used to search HTML documents. XPath uses a path notation (as in URLs) for navigating through the hierarchical structure of an XML document.

The following expressions will select nodes:

Expression	Description
TheNode	Selects all elements with name “TheNode”
/	Selects from the root element
//	Selects descendants of the current element
.	Selects the current element
..	Selects the parent of the current element
@	Selects an attribute of the current element

Below is a sample XML document and some examples of XPath usage.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Movies>
    <Movie>
        <title lang="en">The Avengers</title>
        <Date>
            <Day>20</Day>
            <Month>10</Month>
            <Year>2012</Year>
        </Date>
        <genre>Adventure/Action</genre>
        <rating>9.8</rating>

    </Movie>

    <Movie>
        <title lang="fr">Justice League</title>
        <Date>
            <Day>17</Day>
            <Month>8</Month>
            <Year>2017</Year>
        </Date>
        <genre>Adventure/Thriller</genre>
        <rating>9.3</rating>
    </Movie>
</Movies>
```

Below are some path expressions and the results from them.

Path Expression	Result
Movies	Selects all nodes with the name “Movies”
/Movies	Selects the root element “Movies”
/Movies/Movie	Selects all Movie elements of Movies
//Movies	Selects all Movie elements in the document
Movies//Movie	Selects all Movie elements that are descendants of Movies
@lang	Selects all attributes named lang

Predicates are expressions used to filter nodes selected by a particular path. Predicates are always surrounded by square brackets and give specific information(s) about the element.

Path Expression	Result
/Movies/Movie[1]	Selects the first Movie element
/Movies/Movie[last()]	Selects the last child Movie element
/Movies/Movie[last()-1]	Selects the second to last Movie element
//title[@lang]	Selects all title elements with attribute “lang”
//title[@lang=’en’]	Selects all title elements with attribute lang=en

XPath allows the use of **wildcards** to write more robust path expressions where the use of specific path expressions is either impossible or undesirable

Wildcard	Description
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

2.3.1 XPath operators

XPath defines operators and functions on nodes. An XPath expression returns either a node-set, a string, a Boolean, or a number.

XPath operators can be of different categories according to their property. Following are the different types of XPath operators:

1. **Comparison Operators**

   Comparison operators are used to compare values.

Operator	Description
=	It specifies equals to
!=	It specifies not equals to
<	It specifies less than
>	It specifies greater than
<=	It specifies less than or equals to
>=	It specifies greater than or equals to

2. **Boolean Operators and functions**

   Boolean operators and fucntions are used to check 'and', 'or' & 'not' functionalities.

Operator	Description
and	Operator to check that both conditions must be satisfied.
or	Operator to check that any one of the conditions must be satisfied.
not()	It specifies function to check condition not to be satisfied.

3. Numeric Functions/Operators

A list of **number operators** that are used with XPath expressions:

Operator	Description
+	It is used for addition operation.
-	It is used for subtraction operation.
*****	It is used for multiplication operation.
div	It is used for division operation.
mod	It is used for modulo operation.

A list of **functions on numbers** that are used with XPath expressions:

Functions	Description
ceiling()	It is used to return the smallest integer larger than the value provided.
floor()	It is used to return the largest integer smaller than the value provided.
round()	It is used to return the rounded value to nearest integer.
sum()	It is used to return the sum of two numbers.

4. String Functions

A list of XPath string functions:

Functions	Description
starts-with(str1, str2)	It returns true when str1 starts with the str2.
contains(str1, str2)	It returns true when the str1 contains the str2.
substring(str, offset, length)	It returns a section of the string. The section starts at offset up to the length provided.
substring-before(str1, str2)	It returns the part of str1 up before the first occurrence of str2.
substring-after(str1, str2)	It returns the part of str1 after the first occurrence of str2.
string-length(string)	It returns the length of string in terms of number of characters.
normalize-space(string)	It trims the leading and trailing space from string.
translate(str1, str2, str3)	It returns str1 after any matching characters in str2 have been replaced by the characters in str3.
concat(str1, str2, ...)	It is used to concatenate strings. Function can take more that 2 string parameters.

5. Node Functions

A list of functions on nodes to be used with the XPath expression:

Functions	Description
node()	It is used to select all kinds of nodes.
text()	It is used to select a text node.
name()	It is used to provide the name of the node.
position()	It is used to provide the position of the node.
last()	It is used to select the last node from a set of nodes.

2.3.2 XPath axes

As the location path defines the location of a node using either a relative or absolute path, **axes** are used to identify elements by their relationship like **parent, child, sibling**, etc. Axes are named as such because they refer to axis on which elements are lying relative to an element.

Below is a list of the different axes:

Axis Name	Result
ancestor	These axes indicate all the ancestors relative to the context node, also reaching up to the root node.
child	This indicates the children of the context node.
descendant	This indicates the descendants (children, grandchildren, and their children, …) of the context node. This does NOT indicate the Attribute and Namespace.
following-sibling	This one indicates all the sibling nodes (same parent as the context node) that appear **after or next** to the context node in the HTML DOM (Document Object Model) structure. This does NOT indicate descendent, attribute, and namespace.
parent	This indicates the parent of the context node.
preceding-sibling	his one indicates all the sibling nodes (same parent as context node) that appear **before or previous to** the context node in the HTML DOM structure. This does NOT indicate descendent, attribute, and namespace.
self	This one indicates the context node.

The DOM (Document Object Model) is a programming interface that represents the structure of an HTML or XML document as a tree, allowing scripts to dynamically access, modify, and manipulate its elements and content.

XPath traversal

XPath traversal refers to the navigation between the different nodes in a DOM. Navigation is possible downward (parent to child node), upward (child to parent) or even among siblings with the appropriate use and understanding of the relationship between them.

2.3.3 Absolute and relative XPath

There are two types of XPaths:

1. Absolute XPath
2. Relative XPath

Absolute XPath

The absolute XPath has the complete path starting from the root to the element which we want to identify. The key characteristic of XPath is that it begins with the single forward slash (/).

The major drawback of an absolute XPath is that if there is a change in any node from the root to the expected element, the XPath will no longer be returning the correct node.

```
XPathExpression: /Movies/Movie[1]/Date/Day[1]/text()
```

```
Result:
        20
```

Relative XPath

A relative XPath can start anywhere in the HTML DOM structure or even refer directly to the element that we want to identify.

A relative XPath starts with the // symbol. It is mainly used for automation since even if an element is removed or added in the DOM, the relative XPath is not impacted.

An absolute XPath is lengthy and difficult to maintain (html/body/tagname/…). While a relative XPath is short (//*[@attribute='value']).

```
XPathExpression: //*[text()='20']
```

```
Result:
          20
```

2.4 CSS Selector

Cascading Style Sheets(CSS)

CSS is a language used to describe the look and feel (presentation) of a document (HTML/XML). CSS specifies how the elements will be rendered on screen (or other media). That is, HTML is a markup language and CSS is a style sheet language. HTML and CSS are powerful tools for displaying element on the browser.

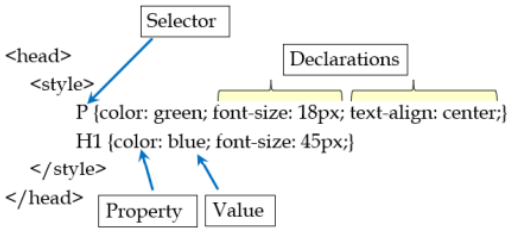
As far as Selenium testing is concerned, CSS is very useful when finding elements for test automation. CSS can be used in HTML Documents in 3 different ways:

1. An external style sheet: each HTML page must include a reference to the external style sheet file inside the element which goes inside the section.
2. An internal style sheet: when a single HTML page is to have a unique style; the styles are defined within the section of the document.
3. An inline style: applies to a specific element and is added directly to the element as an attribute.

When the same element's style is defined by multiple CSS styles, the value from the last read stylesheet will be used. Therefore, the order a style will be used will be defined (starting at the top) as follows:

1. Inline style (as an attribute inside an HTML element)
2. External and internal style sheets defined in the head section
3. The browser default value

A CSS ruleset consists of selector(s) and declaration block(s) as follows:



Each selector refers to an HTML element to style. The declaration blocks consist of one or more declarations separated by semicolons (;). Each declaration includes a CSS property name and a value separated by a colon (:). Declaration blocks are surrounded by curly braces.

CSS selectors are used to find elements in the HTML document based on element name, ID, class, attribute, or other specifiers. Below is a table showing the differences between XPath and CSS selector:

CSS Selector	XPath	Result
div.even	**//div[@class="even"]**	Elements div with an attribute class="even"
#login	**//*[@id="login"]**	An element with id="login"
*****	**//***	All elements
Input	**//input**	All input elements
p,div	**//p | //div**	All p and all div elements
div input	**//div//input**	All input elements inside all div elements
div>input	**//div/input**	All input elements that have the div element as the parent
br + p	**//br/following-sibling::*[1][self::p]**	Selects all p elements that are placed immediately after element br
p ~ br	**//p/following-sibling::br**	Selects all p elements that are placed immediately before element br

CSS selectors also work with attributes as follows:

CSS	Result
[lang]	All elements with the lang attribute
[lang=en]	All elements with the lang attribute of exactly en
f[lang^=en]	All elements with the lang attribute starting with the string en
[lang|=en]	All elements that have the lang attribute equal to en or starting with the string en followed by a hyphen
[lang$=en]	All elements that have the lang attribute ending with the string en
[lang~=en]	All elements that have the lang attribute whose value is a whitespace-separated list of words, one of which is exactly the string en
[lang*=en]	All elements that have lang attribute containing string en

When dealing with form elements, there are a variety of CSS selectors available:

CSS	Result
:checked	Selects all checked elements (for check boxes, radio buttons, and options that are toggled to an on state
:default	Selects any form element that is the default among a group of related elements
:defined	Selects all elements that have been defined
:disabled	Selects all elements that have been disabled
:enabled	Selects all elements that have been enabled
:focus	Selects the element currently with focus
:invalid	Selects any form elements that fail to validate
:optional	Selects form elements which do not have required attribute set
:out-of-range	Selects any input elements whose current value is outside the min and max attributes
:read-only	Selects elements which are not editable by user
:read-write	Selects elements which are editable by user
:required	Selects form elements with required attribute set
:valid	Selects form elements that do validate successfully
:visited	Selects the links that a user has already visited

3 Selenium WebDriver – 240 minutes

Learning Objectives

STF3-1 (K3) Use appropriate logging and reporting mechanisms

STF3-2 (K3) Use hard and soft assertions

STF3-3 (K2) Understand navigation on web browsers

STF3-4 (K3) Use WebDriver commands to change window / tab context in web browsers

STF3-5 (K3) Use WebDriver commands to capture screenshots of web pages

STF3-6 (K4) Differentiate between various strategies to locate GUI elements

STF3-7 (K3) Use WebDriver commands to get state of GUI elements

STF3-8 (K3) Use WebDriver commands to interact with GUI elements

STF3-9 (K3) Use WebDriver commands to interact with user prompts in web browsers

STF3-10 (K2) Understand the new features of Selenium

STF3-11 (K1) Remember the different locators used by Selenium

3.1 Automation Test Logging and Reporting

Automated test scripts are software programs designed to execute commands on the System Under Test (SUT), simulating human actions such as using a keyboard and mouse. Within the Test Automation System (TAS), a mechanism is required to implement the Test Execution Layer. One approach is to write test automation scripts from the ground up, which can be executed like any other Python or Java source code script. Instead of creating these scripts entirely from scratch, existing unit test libraries, such as pytest or TestNG can be used to run tests and report results.

In this syllabus, two distinct test execution libraries for Python and Java programming languages will be used:

Pytest: Pytest is a popular testing framework for Python that supports writing tests. It can be used for scripting unit testing, functional testing, and end-to-end testing in Python projects. Pytest is known for its simplicity, flexibility, and powerful features that enhance test writing and execution.

TestNG: TestNG is a testing framework designed for Java and used to provide coverage from unit testing to integration and end-to-end testing.

Pytest provides a feature called 'fixture' that allows defining reusable setup and teardown code to be shared across multiple tests. Fixtures are used to prepare the environment for test functions, such as initializing objects, setting up databases, or configuring settings before a test runs. After the test is completed, the fixture can also perform any necessary cleanup.
Common Use Cases for Fixtures:

- Setting up a Selenium WebDriver for browser-based testing.
- Connecting to a database and setting up test data.
- Configuring environment variables or application settings.
- Initializing objects that are shared across multiple tests.

Consider the below Python code:

```
import pytest

@pytest.fixture
def sample_fixture():
    # Setup code
    resource = {"key": "value"}
```

```
    yield resource  # Provides the fixture value to the test

    # Teardown code (runs after the test using this fixture)
    resource.clear()

def test_example(sample_fixture):
    # Test using the fixture
    assert sample_fixture["key"] == "value"
```

Below is an explanation of the above:

- **@pytest.fixture**: This decorator marks a function as a fixture.
- **yield**: The code before yield runs as the setup, and the value after yield is what the test function receives as an argument. Any code after yield runs as the teardown, cleaning up after the test.
- **test_example**: This test function uses the sample_fixture fixture. Pytest automatically passes the fixture's value into the test function when it runs.

The equivalent of fixture in Java is TestNG annotations. TestNG annotations are special markers used in the TestNG testing framework for Java to control the flow of test execution. These annotations provide a way to configure methods in your test classes, such as defining which methods should be run as tests, setting up preconditions, or specifying post-test cleanup tasks.
Examples of some TestNG annotations are as below:

@BeforeTest:

- The method under this tag is executed before any test method.

```
@BeforeTest
public void beforeTest() {
    // Setup code for a specific test block
}
```

@Test:

- The methods under this tag are marked as test methods. This is where the test execution happens. The annotations can be configured with a priority parameter as below.

```
@Test(priority = 1)
public void testMethod() {
    // Test code here
}
```

@AfterTest:

- The method under this annotation is executed after all the test methods are executed.

```
@AfterTest
public void afterTest() {
    // Cleanup code for a specific test block
}
```

When a manual tester encounters a failure, they usually have a clear understanding of what went wrong, including the steps leading up to the failure and the data involved. In exploratory testing, testers have a general idea of the execution path and can backtrack to identify issues.
In automation, however, error messages are often insufficient and lack context, making it difficult to pinpoint the cause of a failure. For example, if a failure at step N affects subsequent steps, the log might inaccurately indicate that the failure occurred at step N+1, complicating troubleshooting.
Effective logging can improve this situation by providing detailed information about each test step, including data used and results. This helps quickly identify whether failures are due to the SUT or the automation itself. In critical applications, detailed logging may be required for audits.
While comprehensive logging might seem excessive for small projects, it becomes crucial as automation projects grow. Implementing robust logging from the start is more effective than retrofitting it later.

Python has a very robust set of logging facilities that may be used with WebDriver. At any point in an automated script, the automation engineer may add logging calls to report out any information desired. This can include general information for later tracing the test (e.g., “I am about to click on the XYZ button”), warnings about something that happened that does not rise to the level of a failure (e.g., “Opening file <ABC> took longer than expected.”) or actual errors, raising an exception which triggers end of test events such as cleaning up the environment and moving on to the next test.
The Python logging library has five different levels of messages that may be saved. From lowest to highest levels:

- DEBUG: for diagnosing problems
- INFO: for confirmation that things worked
- WARNING: something unexpected occurred, a potential problem
- ERROR: a serious problem occurred
- CRITICAL: a critical problem occurred

When a log is printed to the console or a file, the message level can be set to one of those five settings (or custom ones can be created) allowing the user to see just the messages desired. For example, if the console is set to level WARNING, the user will not see DEBUG or INFO messages, but will see WARNING, ERROR, and CRITICAL logging messages.
Below Python codes show the implementation of logging.

```
import logging


# Configure the logging system
logging.basicConfig(
    level=logging.DEBUG,  # Set the logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',  # Log message format
    handlers=[
        logging.FileHandler('example.log'),  # Log to a file
        logging.StreamHandler()  # Also log to the console
    ]
)


# Create a logger object
logger = logging.getLogger('exampleLogger')


# Log messages of different severity levels
logger.debug('This is a debug message.')
logger.info('This is an info message.')
logger.warning('This is a warning message.')
logger.error('This is an error message.')
logger.critical('This is a critical message.')
```

Below is output from the above code in console or file.

```
2024-08-24 12:34:56,789 - exampleLogger - DEBUG - This is a debug message.
2024-08-24 12:34:56,789 - exampleLogger - INFO - This is an info message.
2024-08-24 12:34:56,789 - exampleLogger - WARNING - This is a warning message.
2024-08-24 12:34:56,789 - exampleLogger - ERROR - This is an error message.
2024-08-24 12:34:56,789 - exampleLogger - CRITICAL - This is a critical message.
```

There are equivalent logging mechanisms for Java. One of the most popular is Log4j. There are 6 configurable levels of logging for log4j.

- TRACE: Fine-grained informational events that are most useful to debug an application.
- DEBUG: Informational events that are useful for debugging.
- INFO: General information about the application's execution.
- WARN: Potentially harmful situations that might not be errors but which should be monitored.

- ERROR: Error events that indicate a failure in the application.
- FATAL: Severe error events that might cause the application to terminate.

Below Java code shows the implementation of logging using log4j library.

```
import org.apache.log4j.Logger;

public class Example {

    // Initialize the Logger
    private static final Logger logger = Logger.getLogger(Example.class);

    public static void main(String[] args) {
        // Log messages at various levels

        // TRACE level
        logger.trace("This is a TRACE log message.");

        // DEBUG level
        logger.debug("This is a DEBUG log message.");

        // INFO level
        logger.info("This is an INFO log message.");

        // WARN level
        logger.warn("This is a WARN log message.");

        // ERROR level
        logger.error("This is an ERROR log message.");

        // FATAL level
        logger.fatal("This is a FATAL log message.");
    }
}
```

The output for the above code is as below.

```
2024-08-24 12:34:56,789 TRACE Example:10 - This is a TRACE log message.
2024-08-24 12:34:56,789 DEBUG Example:11 - This is a DEBUG log message.
2024-08-24 12:34:56,789 INFO  Example:12 - This is an INFO log message.
2024-08-24 12:34:56,789 WARN  Example:13 - This is a WARN log message.
2024-08-24 12:34:56,789 ERROR Example:14 - This is an ERROR log message.
2024-08-24 12:34:56,789 FATAL Example:15 - This is a FATAL log message.
```

Having logs in console or file is helpful for the automation engineer. But it is also helpful to have a distributable report for a broader audience such as manual testers, developers and managers. The preferred format for this report is html as information can be easily rendered to make the report appealing. There exist libraries that can easily create these reports.

In Python, the most common html report generation library is pytest html. The command below could be used to generate an html report:
pytest Exercises.py --html=report.html

In Java, the most popular html report generation library is ExtentReport. SparkReporter is a HTML format ExtentReport which is widely used. The below libraries are to be imported to use the reporting libraries in Java.

```
Import com.aventstack.extentreports.*;
import com.aventstack.extentreports.reporter.ExtentSparkReporter;
```

3.1.1 Test Assertions

Test assertions are crucial for verifying that an SUT behaves as expected. They are statements that check if a condition is true at a specific point during a test. If the condition is false, an assertion failure is reported, indicating a potential issue in the application.

There are two types of assertions needed in tests.

Hard Assertions:

- Definition: Hard assertions stop the test execution immediately when they fail. They are used to enforce critical conditions that must be met for the test to proceed.
- Use Case: Suitable for conditions that are essential for the test's integrity. For example, if a login page fails to load, subsequent steps should not be executed.
- Example: In many testing frameworks, hard assertions throw an exception when the condition fails, halting further execution.

Soft Assertions:

- Definition: Soft assertions allow the test to continue even if they fail. They are useful for collecting multiple issues in a single test run, providing a comprehensive view of potential problems.
- Use Case: Ideal for non-critical checks where the test can still provide valuable feedback despite some failures. For instance, validating several form fields where individual errors do not prevent overall form submission.
- Example: Soft assertions collect failures and report them after the test completes, allowing all specified checks to be executed.

In Python,soft assertion can be implemented as below.

```
assert driver.current_url == "https://www.saucedemo.com/", "URL did not match."
```

To make an assertion a hard one, we need to add the assertion in a try and catch construct so that execution is then halted. In Java the soft/hard assertions are differentiated by the keywords SoftAssert and Assert, while in python it is done using assert keyword (for soft assert) and try/catch block with assert for hard assert.

In Java, assertion from the TestNG library can be used. The below libraries need to be imported.
import org.testng.annotations.Test;
import org.testng.asserts.SoftAssert;

The implementation of a hard assertion in Java TestNG library is as below.

```
//Verify if website title is correct using hard assert

String ActualTitle = webdriver.getTitle();
String ExpectedTitle = "Swag Labs";
Assert.assertEquals(ActualTitle, ExpectedTitle);
```

The implementation of a soft assertion in Java TestNG library is as below.

```
//Verify if product filter is displayed using soft assert

SoftAssert softassert = new SoftAssert();
WebElement prodfilter = webdriver.findElement(By.className("product_sort_container"));
softassert.assertEquals(true, prodfilter.isDisplayed());
```

Below is a non-exhaustive list of useful information that can appear in a test report for an automated test execution:

- Total number of test cases
- Number of passed test cases
- Number of failed test cases

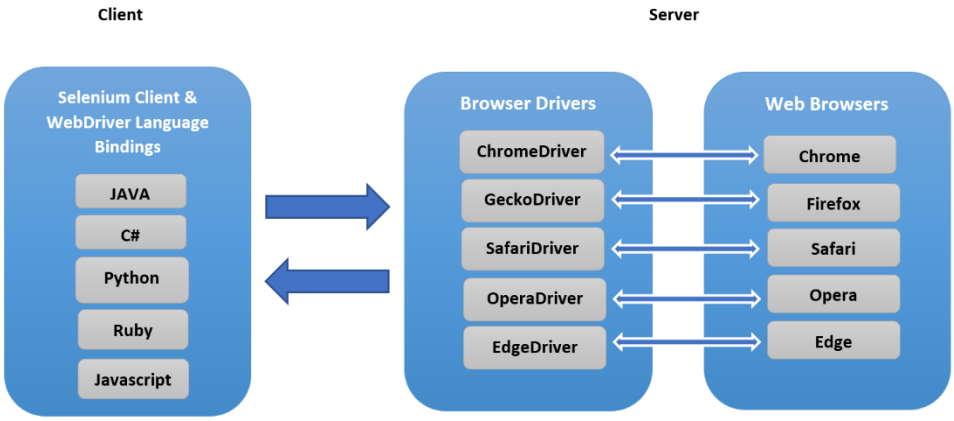
- Number of skipped test cases: The count of tests that were not executed, often due to dependencies or configuration issues.
- Pass rate: The percentage of tests that passed (e.g., Passed Tests / Total Tests * 100).
- Fail rate: The percentage of tests that failed (e.g., Failed Tests / Total Tests * 100).
- Test start time
- Test end time
- Total execution time: The total time taken to execute all test cases.
- Test case execution time: The time taken to execute each test case
- Test environment: Information about the environment where tests were executed, such as operating system, browser version, and hardware specs.
- Build version: The specific version of the application under test.
- Test data used: Description of the data sets used during the test execution.
- Logs and screenshots: Attachments or links to logs, screenshots, or videos captured during test execution, especially for failed tests.

A test report from a TAS is expected to be as detailed as possible so that a manual tester can confirm if discrepancy from actual result and expected result is a false positive or not. The information can also be used for a root cause analysis. For example, the time stamps on the report could be used to search the activities on the SUT log to explain a failed test case.

3.2 Managing the Selenium WebDriver

Cross-browser testing and concurrent testing are both supported by the Selenium WebDriver architecture. For code compilation, Selenium WebDriver offers integration with several programming languages (such as Java, Python, Ruby, C# or JavaScript) and build automation tools (such as Maven or ANT).

The architectural representation of Selenium 4 WebDriver is shown below:



The Selenium WebDriver architecture consists of the following three elements:

- Selenium Client Library / Language Bindings
- Browser Drivers
- Browsers

Selenium Client Library / Language Bindings

Multiple programming languages, including Java, C# and Python are supported by Selenium 4. Language bindings were created by Selenium developers to accommodate a variety of languages. Once the preferred language is chosen, the language binding library can be added to the automation development project.
If Python programming language is used, the language binding can be added using the pip command.
If Java is used, the language binding can be imported as an external JAR file (Java Archive) or, it can be added as dependencies if a Maven or a Gradle project is used for development.

Browser Drivers

Each of the Selenium-supported browsers has its own implementation of the W3C standard because Selenium supports a wide range of browsers. As a result, browser-specific binaries are readily available; these binaries are particular to the browser and shield the user from the implementation logic. The client libraries and the browser binaries are now directly interacting with each other as opposed to the previous versions of Selenium where the JSON Wire Protocol was in between to make the interface between the two.

Browsers

Only browsers that are locally installed, either on the local workstation or the server machines, will be allowed to conduct tests on Selenium. Installing a browser is therefore mandatory.

Before WebDriver could be used, the necessary libraries are to be imported.

Python provides an installation of library packages though the ‘pip’ command. To use Selenium 4, minimum version of Python 3.7 will be required.

For example, after Python is installed and the necessary environment variables are set, the command below will install Selenium 4 on the machine.

```
pip install selenium==4.5.0
```

The below command will install ChromeDriver on the machine:

```
pip install ChromeDriver
```

Alternatively, the binary of the library can be referred to through its path. The variable ‘binary_path’ stores the path to the chrome driver executable.

```
from ChromeDriver_py import binary_path
```

In Java, there are dependency management and build solution tools. Maven and Gradle are the most common for Selenium.

The Maven solution introduces a project object model file referred to POM file which is an XML structure file. Every time a change is made to the project code, it updates the build status and continuously maintains and monitors the framework components and build. If there are no compilation problems with the framework, it provides a "build success" message; otherwise, it provides a "build failure" message.

A minimum version of Java 8 is recommended to install Maven dependency management. Environment variables need to be set. Once done, dependencies can be added to the ‘pom.xml’ file.

After adding the dependency for Selenium, the Pom.xml file will look like this:

```
<dependencies>
   <!-- more dependencies ... -->
   <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>4.5.0</version>
   </dependency>
   <!-- more dependencies ... -->
</dependencies>
```

Similar to how it is done with Maven, adding dependencies to a Gradle project (in build.gradle file) is simple as shown below:

```
dependencies {

implementation group: 'org.seleniumhq.selenium', name: 'selenium-java', version: '4.5.0'}
```

Gradle automatically executes all of the tests that it finds, which it does by looking at the test classes that have been compiled. Gradle searches for and executes all methods marked with the @Test annotation when useTestNG() is specified.
Once the libraries are all set, the WebDriver can be instantiated to run tests. The initialization is done based on the browser controller we want to invoke.

To initialize the WebDriver in Python, after the necessary libraries are installed, the WebDriver class need to be imported from Selenium into the solution.

The below code will invoke a Google Chrome browser controller.

```
from selenium import WebDriver

driver = WebDriver.Chrome()
```

Similarly for Firefox browser, the below code can be used.

```
from selenium import WebDriver

driver = WebDriver.Firefox()
```

If specific browser configurations are needed, then object ‘options’ can be used as shown below:

```
from selenium.WebDriver.firefox.options import Options as FirefoxOptions
options = FirefoxOptions()
options.browser_version = '92'
options.add_argument("--headless")
options.platform_name = 'Windows 10'
driver = WebDriver.Firefox(options=options)
```

In the above Python code, we are declaring ‘options’ as configuration for a Firefox browser controller, setting the version of the browser to load as 92, in headless mode for Microsoft Windows 10.

Listed below are the Options objects to be used going forward for defining browser-specific capabilities:

- Firefox – FirefoxOptions
- Chrome – ChromeOptions
- Internet Explorer(IE) – InternetExplorerOptions
- Microsoft Edge – EdgeOptions
- Safari – SafariOptions

Similarly, Java, the following libraries need to be invoked for the WebDriver initialization are:

1. **org.openqa.selenium.*–** contains the WebDriver class needed to instantiate a new browser loaded with a specific driver
2. **org.openqa.selenium.firefox.FirefoxDriver** – contains the FirefoxDriver class needed to instantiate a Firefox-specific driver onto the browser instantiated by the WebDriver class.

OR

org.openqa.selenium.chrome.ChromeDriver – contains the ChromeDriver class needed to instantiate a Chrome-specific driver onto the browser instantiated by the WebDriver class.

OR

org.openqa.selenium.edge.EdgeDriver – contains the EdgeDriver class needed to instantiate an Edge-specific driver onto the browser instantiated by the WebDriver class.

OR

org.openqa.selenium.safari.SafariDriver – contains the SafariDriver class needed to instantiate a Safari-specific driver onto the browser instantiated by the WebDriver class.

Note that org.openga.selenium.***** will import all packages in the library which is not always the best way to move forward. It is a good coding principle, not to inject dependencies and libraries that are not needed into the code. This eases long term-code maintainability. Therefore, if more specific packages are needed, it may be better to just import the required ones.

For example:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless");
options.setPlatformName("Windows 10");
options.setBrowserVersion("92");
driver = new ChromeDriver(options);
```

In the above Java code, we are declaring ‘options’ as configuration for a Chrome browser controller, setting the version of the browser to load as 92, in headless mode for Microsoft Windows 10.

3.2.1 WebDriver Interactions

Once the WebDriver is initialized, we can start navigating to a website, get the website title or URL, get the page source. and close the driver. To do so, browser controllers provide various methods for the Selenium client to interact with the browsers.

Some of the most popular Browser controllers for Selenium WebDriver are listed below. The commands below work for both Python (without the ending semicolon) and Java (with the ending semicolon).

1) Get Command

   This method loads a fresh tab in the browser and load a new webpage. Takes a String as an input and gives no output.

   The respective command to load a fresh web page is:

```
driver.get(URL);
```

2) Get Title Command

The title of the current page is retrieved using this method. Returns a String value and takes nothing as a parameter.

The respective command to retrieve the title of the currently displayed page:

```
driver.getTitle();
```

3) Get Current URL Command

This method retrieves the string that corresponds to the currently open URL in the browser. Returns a String value and takes nothing as a parameter.

The respective command to retrieve the string that represents the current URL is:

```
driver.getCurrentUrl();
```

4) Get Page Source Command

The page's source code is returned by this method. Returns a String value and takes nothing as a parameter.

The respective command to obtain the current web page's source code is:

```
driver.getPageSource();
```

5) Get previous page

This method is useful for simulating the browser's back button in test scripts.

```
driver.back();
```

6) Get next page

This method is useful when you need to test navigation flows that involve moving forward in the browser's history.

```
driver.forward();
```

7) Refresh page

This method is used to refresh the current page, effectively simulating the browser's refresh or reload button.

```
Python: driver.refresh()
Java: driver.navigate().refresh();
```

8) Full Screen, Maximize and Minimize webpage

This method is used to maximize, minimize or full screen a webpage.

```
Python:
	# Maximize the browser window
	driver.maximize_window()

	# Minimizes the window of current browsing context
	driver.minimize_window()

	# Fills the entire screen
	driver.fullscreen_window()

Java:
	# Maximize the browser window
	driver.manage().window().maximize();

	# Minimizes the window of current browsing context
	driver.manage().window().minimize();

	# Fills the entire screen
	driver.manage().window().fullscreen();
```

9) Close Command

This method closes the window that WebDriver is currently in control of.

Returns nothing and accepts nothing as an argument.

The respective command to close the browser window is:

```
driver.close();
```

10) Quit Command

```
quit(): void
```

All windows that the WebDriver has opened are closed by this method. Returns nothing and accepts nothing as an argument.

The respective command to shut down all windows is:

```
driver.quit();
```

3.2.2 Windows and Tabs Management

Managing multiple browsers and browser tabs (Google Chrome in this example) are at times very important for some tests. Some pages have URL links that open up in new tabs only. There is a need to be able to swap control to different frame contexts, browser windows, or tabs. Selenium 4 comes with a new API interface: newWindow() that allows users to create and switch to a new window/tab without creating a new WebDriver object.

Sample code snippet in Python to open a new window

```
from selenium import webdriver
from selenium.webdriver.common.window import WindowTypes

# Initialize the WebDriver (e.g., using Chrome)
driver = webdriver.Chrome()

# Open Google homepage
driver.get("https://www.google.com/")

# Open a new window and switch to it
driver.switch_to.new_window(WindowTypes.WINDOW)

# Open Alliance for Qualification homepage in the newly opened window
driver.get("https://allianceforqualification.com/")
```

Sample code snippet in Python to open a new tab within the same window

```
from selenium import webdriver
from selenium.webdriver.common.window import WindowTypes

# Initialize the WebDriver (e.g., using Chrome)
driver = webdriver.Chrome()

# Open Google homepage
driver.get("https://www.google.com/")

# Open a new tab in the existing window and switch to it
driver.switch_to.new_window(WindowTypes.TAB)

# Open A4Q homepage in the newly opened tab
driver.get("https://allianceforqualification.com/")
```

Sample code snippet in Python to switch content to a frame by index, by name or ID or by WebElement. Then the control is given back to the main context.

```
import org.openqa.selenium.By; from selenium import webdriver
from selenium.webdriver.common.by import By
```

```
# Initialize the WebDriver (e.g., using Chrome)
driver = webdriver.Chrome()

# Navigate to a website with a frame
driver.get("https://example.com")

# Switch to a frame by index
driver.switch_to.frame(0)

# Switch to a frame by name or ID
driver.switch_to.frame("frameNameOrId")

# Switch to a frame by WebElement
frame_element = driver.find_element(By.XPATH, "//iframe[@id='frameID']")
driver.switch_to.frame(frame_element)

# Perform actions within the frame
# ...

# Switch back to the default content (the main page)
driver.switch_to.default_content()

# Close the browser
driver.quit()
```

Sample code snippet in Java to open a new window

```
driver.get("https://www.google.com/");

// Opens a new window and switches to new window
driver.switchTo().newWindow(WindowType.WINDOW);

// Opens A4Q homepage in the newly opened window
driver.navigate().to("https://allianceforqualification.com/");
```

Sample code snippet in Java to open a new tab within the same window

```
driver.get("https://www.google.com/");

// Opens a new tab in existing window
driver.switchTo().newWindow(WindowType.TAB);

// Opens A4Q homepage in the newly opened tab
driver.navigate().to("https://allianceforqualification.com/");
```

Sample code snippet in Java to switch content to a frame by index, by name or ID or by WebElement. Then the control is given back to the main context.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class SwitchToFrameExample {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();

        // Navigate to a website with a frame
```

```
        driver.get("https://example.com");

        // Switch to a frame by index
        driver.switchTo().frame(0);

        // Switch to a frame by name or ID
        driver.switchTo().frame("frameNameOrId");

        // Switch to a frame by WebElement
        WebElement frameElement =
driver.findElement(By.xpath("//iframe[@id='frameID']"));
        driver.switchTo().frame(frameElement);

        // Perform actions within the frame
        // ...

        // Switch back to the default content (the main page)
        driver.switchTo().defaultContent();

        // Close the browser
        driver.quit();
    }
}
```

3.2.3 Screen captures

In Selenium 4, new methods allow for capturing screenshots of a specific web element or section of a page, in addition to page capture.

The getScreenshotAs() method is used to get screenshot of element specified by the locator.

Sample code snippet for element level or section level screen capture in Python

```
from selenium import webdriver
from selenium.webdriver.common.by import By
driver = webdriver.Chrome()
driver.get("http://www.example.com")
welement = driver.find_element(By.CSS_SELECTOR, 'h1')
welement.screenshot('./image.png')
driver.quit()
```

Sample code snippet for page level screen capture in Python

```
from selenium import webdriver
from selenium.webdriver.common.by import By
# For file operations
import os

driver = webdriver.Chrome()
driver.get("http://www.example.com")

# Take a screenshot and save it
screenshot = driver.get_screenshot_as_file("ViewPage.png")
driver.quit()
```

Sample code snippet for element level or section level screen capture in Java

```
public class SeleniumelementTakeScreenshot
{
  public static void main(String args[]) throws IOException
  {
    WebDriver driver = new ChromeDriver();
    driver.get("https://www.example.com");
    WebElement element = driver.findElement(By.cssSelector("h1"));
    File scrFile = element.getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new File("./image.png "));
    driver.quit();
  }
}
```

Sample code snippet for page level screen capture in Java

```
public class SeleniumelementTakeScreenshot
{
  public static void main(String args[]) throws IOException
  {
    WebDriver driver = new ChromeDriver();
    driver.get("https://www.example.com");
    File srcFile =
((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(srcFile, new File("ViewPage.png"));
    driver.quit();
  }
}
```

3.3 Web Elements Interactions

In Selenium WebDriver, we can find elements on a web page using various methods. These methods can be used to locate a single element, or multiple elements based on different criteria. Here are the different methods to find elements in both Java and Python using Selenium 4.

Sample code snippet for finding single element in Python.

```
1. By ID
        element = driver.find_element(By.ID, "elementId")
2. By Name
        element = driver.find_element(By.NAME, "elementName")
3. By Class Name
        element = driver.find_element(By.CLASS_NAME, "className")
4. By Tag Name
        element = driver.find_element(By.TAG_NAME, "tagName")
5. By Link Text
        element = driver.find_element(By.LINK_TEXT, "Link Text")
6. By Partial Link Text
        element = driver.find_element(By.PARTIAL_LINK_TEXT, "Partial Link Text")
7. By CSS Selector
        element = driver.find_element(By.CSS_SELECTOR, "cssSelector")
8. By XPath
        element = driver.find_element(By.XPATH, "xpathExpression")
```

Sample code snippet for finding multiple elements in Python.

```
1. By ID
     elements = driver.find_elements(By.ID, "elementId")
2. By Name
     elements = driver.find_elements(By.NAME, "elementName")
3. By Class Name
     elements = driver.find_elements(By.CLASS_NAME, "className")
4. By Tag Name
     elements = driver.find_elements(By.TAG_NAME, "tagName")
5. By CSS Selector
     elements = driver.find_elements(By.CSS_SELECTOR, "cssSelector")
6. By XPath
     elements = driver.find_elements(By.XPATH, "xpathExpression")
```

Sample code snippet for finding single element in Java.

```
1. By ID
      WebElement element = driver.findElement(By.id("elementId"));
2. By Name
      WebElement element = driver.findElement(By.name("elementName"));
3. By Class Name
      WebElement element = driver.findElement(By.className("className"));
4. By Tag Name
      WebElement element = driver.findElement(By.tagName("tagName"));
5. By Link Text
      WebElement element = driver.findElement(By.linkText("Link Text"));
6. By Partial Link Text
      WebElement element = driver.findElement(By.partialLinkText("Partial Link
      Text"));
7. By CSS Selector
      WebElement element = driver.findElement(By.cssSelector("cssSelector"));
8. By XPath
      WebElement element = driver.findElement(By.xpath("xpathExpression"));
```

Sample code snippet for finding multiple elements in Java.

```
1. By ID
      List<WebElement> elements = driver.findElements(By.id("elementId"));
2. By Name
      List<WebElement> elements = driver.findElements(By.name("elementName"));
3. By Class Name
      List<WebElement> elements = driver.findElements(By.className("className"));
4. By Tag Name
      List<WebElement> elements = driver.findElements(By.tagName("tagName"));
5. By CSS Selector
      List<WebElement> elements =
      driver.findElements(By.cssSelector("cssSelector"));
6. By XPath
      List<WebElement> elements =
      driver.findElements(By.xpath("xpathExpression"));
```

Relative or friendly locators in Selenium 4

Selenium 4 also brings an easy way of locating elements with the inclusion of relative locators. This means testers can now locate specific web elements using intuitive terms that are often used by users like:

- **toLeftOf():** Find the element which is to the left of a specified element.
- **toRightOf():** Find the element which is to the right of the specified element.
- **above():** Find the element which is above with respect to the specified element.
- **below():** Find the element which is below with respect to the specified element.
- **near():** Find the element which is at most 50 pixels (default value) far away from the specified element. The pixel distance value can be modified.

The introduction of friendly locators in Selenium 4 helps locate web elements based on the visual / UI location relative to other DOM elements.

Once we can locate the desired elements, we need to ensure that the elements are in the expected condition to be able to act on it. In Selenium WebDriver, expected conditions are used in conjunction with WebDriver's wait command to wait for specific conditions to be met before proceeding with actions. These expected conditions help the tester to manage and synchronize the tests with dynamic content and elements on web pages, making the tests more reliable and less prone to timing-related issues.

Here is a list of commonly used expected conditions provided by Selenium:

No.	Python	Java	Description
1	title_is(title)	titleIs(String title)	Waits for the title of the page to be exactly equal to the given title.

2	title_contains(title)	titleContains(String title)	Waits for the title of the page to contain the given substring.
3	presence_of_element_located(locator)	presenceOfElementLocated(By locator)	Waits for an element to be present in the DOM of the page.
4	visibility_of_element_located(locator)	visibilityOfElementLocated(By locator)	Waits for an element to be visible on the page.
5	visibility_of(element)	visibilityOf(WebElement element)	Waits for the element to be visible.
6	presence_of_all_elements_located(locator)	presenceOfAllElementsLocatedBy(By locator)	Waits for all elements matching the locator to be present in the DOM
7	text_to_be_present_in_element(locator, text)	textToBePresentInElement(By locator, String text)	Waits for the text to be present in the specified element.
8	text_to_be_present_in_element_value(locator, text	textToBePresentInElementValue(By locator, String text)	Waits for the text to be present in the value attribute of the specified element.
9	attribute_to_be(locator, attribute, value)	attributeToBe(By locator, String attribute, String value)	Waits for the attribute of an element to have a specific value.
10	element_to_be_clickable(locator)	elementToBeClickable(By locator)	Waits for an element to be clickable.
11	staleness_of(element)	stalenessOf(WebElement element)	Waits for the element to become stale (i.e., no longer present in the DOM).
12	alert_is_present()	alertIsPresent()	Waits for an alert to be present.
13	element_to_be_selected(element)	elementToBeSelected(WebElement element)	Waits for an element to be selected (e.g., a checkbox or radio button).
14	frame_to_be_available_and_switch_to_it(locator)	frameToBeAvailableAndSwitchToIt(By locator)	Waits for a frame to be available and then switches to it.
15	invisibility_of_element_located(locator)	invisibilityOfElementLocated(By locator)	Waits for an element to be invisible.
16	invisibility_of_element(element)	invisibilityOf(WebElement element)	Waits for the specific element to be invisible.
17	element_located_to_be_selected(locator)	elementLocatedToBeSelected(By locator)	Waits for an element located by the given locator to be selected.

Sample code snippet for implementation of expected condition

```
In Python:
from selenium.webdriver.support import expected_conditions as EC
# Example of waiting for an element to be visible
WebDriverWait(driver, 10).until(EC.visibility_of_element_located((By.ID,
"elementId")))


In Java:
// Example of waiting for an element to be visible
WebElement element =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("elementId")));
```

We can now use some additional web element assessor methods to ensure correct interactions are executed at the right time. Below are some common properties and assessor methods.

Python	Java	Description
text	getText()	Returns the visible (inner) text of the element.
get_attribute(attribute_name)	getAttribute(String attributeName)	Returns the value of the specified attribute.
tag_name	getTagName()	Returns the tag name of the element.
value_of_css_property(property_name)	getCssValue(String propertyName)	Returns the value of the specified CSS property.
size	getSize()	Returns the size of the element as a dictionary with width and height.
location	getLocation()	Returns the location of the element as a dictionary with x and y coordinates.
is_displayed()	isDisplayed()	Returns True if the element is visible, otherwise False.
is_enabled()	isEnabled()	Returns True if the element is enabled, otherwise False.
is_selected()	isSelected()	Returns True if the element is selected (for checkboxes, radio buttons, etc.), otherwise False.

The below methods are the basic interactions that can be done on a web element.

When a click() operation is executed, the automation solution needs one parameter which defines where to click.
When a sendKeys() operation is executed, the automation solution needs two parameters which define where to enter the text and what text to enter.
The web element defines the 'where' parameter and the parameter specified in the method defines the 'what' parameter.

The table below summarizes the most common methods available for a web element.

Python Method	Java Method	Description
click()	click()	Clicks on the element
send_keys(string)	sendKeys(string)	Enter the string or characters in the specified element
clear()	clear()	Clears the text present in the element
text	getText()	Retrieves the text present in the element
get_attribute(string)	getAttribute(string)	Retrieves the value contained in the attribute name specified from the DOM for the element

The below methods are the basic interactions that can be done on a web element.
When a click() operation is executed, the automation solution needs one parameter which defines where to click.

Consider the following DOM segment for an element:

```
<a href="https://www.alliance4qualification.info/" id="link" />
```

In Python, the getAttribute() method will be used as below:

```
element = driver.find_element(By.ID, "link")
element.get_attribute('href')
```

In Java, the getAttribute() method will be used as below:

```
element = driver.findElement(By.id("link"));
element.getAttribute ('href')
```

The output of the above instructions would then be:
https://www.alliance4qualification.info/

3.3.1 Browser Alert Management

In web automation and testing, there are generally three types of browser alerts (also known as JavaScript pop-ups) that we might encounter.

1) Alert: A pop-up with a message and an "OK" button to acknowledge the information.
2) Confirm: A pop-up with a message, "OK," and "Cancel" buttons to confirm or reject an action.
3) Prompt: A pop-up with a message, an input field, and "OK" and "Cancel" buttons to collect user input.

These alerts appear as modal windows which need to be handled before interactions with the browser elements can be continued. In order for the TAS to interact with the alert modal windows, the context needs to be switched. Below are codes snippets on handling alerts.

Handling browser alert in Python

```
alert = driver.switch_to.alert
print(alert.text)
alert.accept()
```

Handling browser alert in Java

```
Alert alert = driver.switchTo().alert();
System.out.println(alert.getText());
alert.accept();
```

The following alert management interfaces are possible using Selenium 4:

- **Accept an alert:** accept()
- **Dismiss an alert:** dismiss()
- **Get text from an alert:** getText() (Java) / text (Python)
- **Send text to a prompt:** sendKeys(String text) (Java) / send_keys("text") (Python)

4 Maintainability of TAS and Test Scripts – 150 minutes

Learning Objectives

STF4-1 (K2) Understand which factors support and affect the maintainability of test scripts

STF4-2 (K3) Use appropriate wait mechanisms

STF4-3 (K4) Analyze GUI of SUT and use Page Objects to make its abstractions

STF4-4 (K4) Analyze test scripts and apply Keyword Driven Testing principles to building test scripts

4.1 Test script maintenance

At its fundamental level, a manual test comprises a series of abstract instructions that only gain value when executed by a manual tester. While the inclusion of data and expected results is necessary, the essence lies in the sequence of steps to be performed. The manual tester contributes context and refinement to these abstract instructions, thereby enabling the successful execution of even the most complex tests.

For instance, when a manual test instructs the tester to click a button, the action is performed with minimal caution. The tester does not need to consciously verify whether the control is visible, enabled, or appropriate for the test to be done. Nevertheless, these considerations are implicitly made. If the control is not visible, the tester may attempt to make it so. If it is not enabled, the tester will not use the control and will instead investigate the reason why it is disabled and seek a resolution. Once the control is accessible, the tester can proceed accurately, and should any issue arise, they are capable of identifying the cause, documenting it in a defect report, and potentially revising the manual test procedure.

In contrast, every automation tool, regardless of its sophistication or cost, operates without the context and judgment that human testers inherently possess. However, automation engineers, with their human intellect, can enhance these tools by incorporating context and reasoning into the automated scripts through deliberate programming.

However, the more we try to program intelligence into automated scripts, the more complex they become, increasing the likelihood of failures due to this inherent complexity.
Writing automation scripts for testing is often compared to trying to shoot down a bullet with another bullet, highlighting the complexity involved. Complexity, though, is a double-edged sword. We need to incorporate intelligence into our scripts to better simulate a human tester's actions, especially given the growing complexity of the systems under test (SUTs). However, as we add more complexity to our automation, we also increase the chances of encountering automation failures.
No matter how skilled we are as automation engineers, there are limits to what conditions we can manage through scripts. A manual tester might investigate why a control is not visible, whereas an automation engineer might not be able to; instead, we are often limited to coding the script to wait a finite amount of time, hoping the issue resolves itself. The same challenge applies to whether a control is enabled.
An automation engineer can ensure that a control is in the correct state before interacting with it. If not, the script can wait a set amount of time, and if the control remains unusable, log the error and

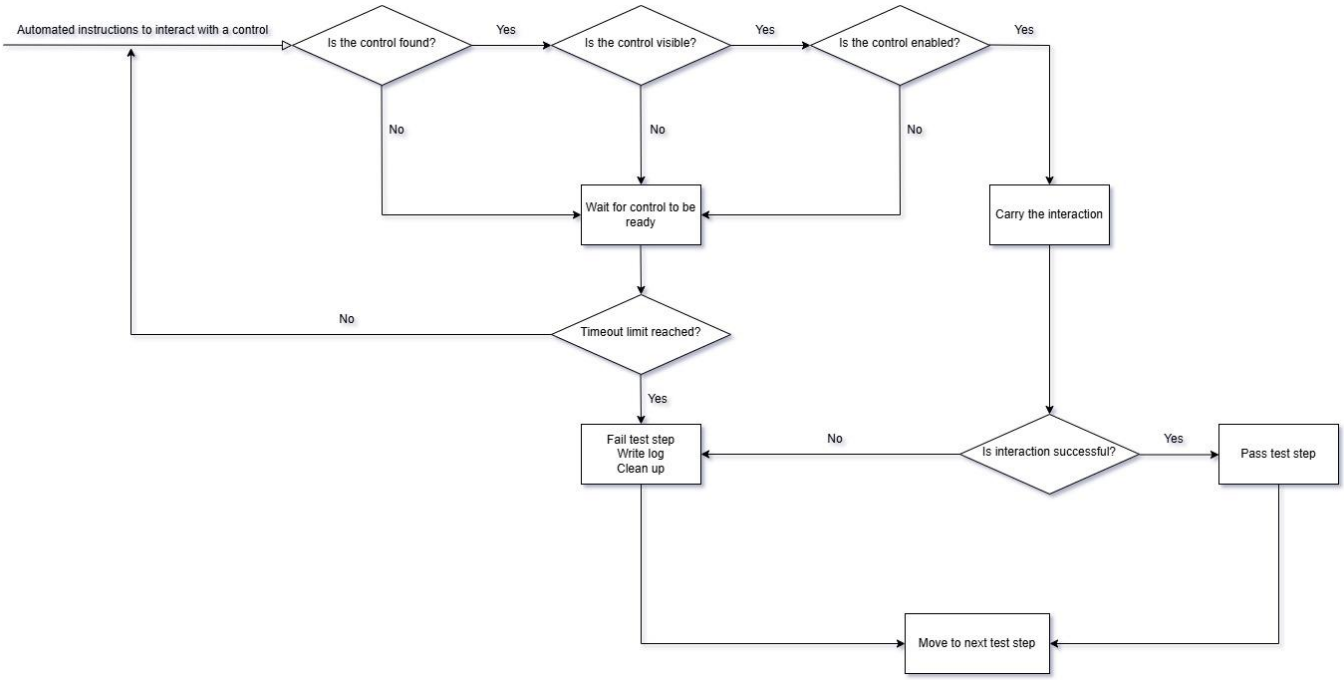
exit the script gracefully to proceed with the next test. If the control can be accessed, we can verify whether it behaves as expected after manipulation. If there is an issue, we can log a detailed message to help troubleshoot more effectively.

To simplify our work, we aim to build intelligence into reusable functions rather than embedding it in every individual script. In other words, we shift the intelligence to the Test Automation Architecture (TAA) and/or Test Automation Framework (TAF), removing it from the script level.

By moving more intelligence to higher levels, scripts become more maintainable and scalable. While failures in this centralized code might affect multiple scripts, fixing it at a single point of contact simplifies overall maintenance.

We provided examples of creating these reusable functions. However, the code we described is not quite ready for production automation. Skilled automation engineers tend to develop these aggregate functions, often referred to as wrapper functions. For instance, consider a wrapper function designed to click a checkbox. The goal is to create a function that will mirror the thought processes and actions of a manual tester.

The arguments for this wrapper function might include the checkbox to interact with, the desired final state (checked or unchecked), and possibly the maximum time we are willing to wait if the checkbox is not immediately ready. The function would then follow a logical sequence of tasks based on these inputs.



Significant effort should be dedicated to the "failed" block. The log statements must be detailed enough to minimize the time spent troubleshooting when a test fails. The cleanup functionality should ensure that the test suite can seamlessly proceed to the subsequent tests.

It is often said that the most important test to consider is the next one. If you can always run the next test, you can complete the entire suite, regardless of how many tests fail during execution.

Ultimately, testers aim to identify where the system under test (SUT) fails to function correctly (and to build confidence when it appears to work correctly). If our automation is effective, failures should be expected; we must collect the failure information and move on to the next test.

Remember, each test is designed to examine the SUT, its environment, and its usage to reveal what we do not already know. A test that does not provide new insights is not valuable, whether it is automated or not.

While essential, building other functions to enhance automation is also crucial. Any code that needs to be called repeatedly in the automation process should be turned into a function. This practice aligns with good development principles such as functional decomposition and refactoring. By creating libraries of functions, you equip yourself and your teammates with valuable toolkits. Whenever possible, move everything out of the actual scripts into these reusable libraries.

Building these functions requires time and resources, but it is one of the best investments you can make if you want to succeed in test automation. When testing in a browser, avoid using absolute paths when locating web elements with XPath and CSS Selectors. Changes to the HTML structure are likely to cause absolute paths to change, breaking the automation run. While relative paths can also break, they tend to be more resilient and therefore require less maintenance.

It is important to discuss your automation strategies with the developers in your organization. Share your challenges with them; there are many ways they can write their code to prevent frequent breaks in the test automation when it comes to interacting directly with the HTML.

Use meaningful global names (variables, constants, function names, etc.) to enhance code readability, which in turn makes maintaining the automation code easier. Easier maintenance often leads to fewer regression defects when changes are made. Taking a few extra seconds to come up with descriptive names can save you hours of work later.

Comments are critical so it is important to include plenty of meaningful comments. Many test automation engineers assume they will remember why they wrote certain pieces of code the way they did or think that automation is somehow different from "real" code. You can easily forget the clever solution you implemented from one day to the next. Do not reinvent the wheel. As your automation project gains success, others will start working with your code, so make it easy for them to understand.
Create dedicated test accounts and fixtures specifically for test automation. Test automation needs consistency when it comes to the data used.

Create sufficient data within these accounts to make them resemble real user accounts. Do not overlook the importance of incorporating different personas. If your SUT supports various types of users, these personas should be represented in your test accounts.

It is crucial to take logging seriously. If automation efforts are successful, management will likely want more, and scalability becomes key. One way to make automation more scalable is to implement good logging practices from the outset. You will not retrofit hundreds or even thousands of scripts with proper logging later.

Python and Java offers a robust set of logging tools, or you can develop your own logging framework. Comprehensive logs can significantly reduce the time needed for troubleshooting when failures occur. Consider the needs of the organization—if the SUT is mission-critical or safety-critical, your logs may need to be thorough enough to meet audit requirements.
Try to emulate the thought processes of manual testers.

Manage the files created carefully. Use consistent naming conventions and save files in consistent directories. Consider creating a folder with a timestamp in its name and storing all files from a run there. If multiple machines are running the automated test or if testing in different environments, consider adding the workstation name or environment names to the folder names.
Including timestamps in file names is a good practice to ensure the TAS does not overwrite previous test results.

Maintain consistency in file naming. Collaborate with other automation engineers to establish a set of standards and naming conventions. Ensure these guidelines are taught to new team members. It does not take long for chaos to ensue when everyone follows their own rules in creating persistent artifacts.

Historically, test automation engineers have been seen as amateurs of development—individuals who like to do things their way and push boundaries. However, the investment in test automation is substantial, both in terms of resources and time. To ensure that this investment pays off, it is important to adopt minimum standards and guidelines.

4.2 Wait mechanism

When a manual tester runs a test, the concept of waiting is not usually a significant concern. For example, if the tester opens a file and it opens within a reasonable timeframe (as judged by the tester), no further thought is given to it. Whenever a manual tester clicks on a control or interacts with the SUT, they have an implicit sense of timing in their mind. If something seems to take too long, they are likely to rerun the test, this time paying close attention to the timing. However, what a tester will never do is sit for hours, patiently waiting for a particular action to complete.

Timing is one of those contextual elements that must be understood. For instance, if a tester is opening a very small file—just a few hundred bytes—and it doesn't open instantly, the tester might become concerned and consider filing an incident report. On the other hand, if the same tester tries to open a two-gigabyte file and it takes thirty seconds to open, the tester might not be surprised at all. If the tester receives an informational message indicating that the file will be slow to open, they might simply dismiss the message and continue waiting without a second thought. Therefore, the context matters.

In test automation, regardless of the tool used, context is largely absent. Typically, the tool has a predefined amount of time it is willing to wait for an action to complete. If the expected action does not occur within that timeframe, the tool records it as a failure and moves on to the next step.

For example, if the tool is set to wait for 3 seconds, an action that occurs in 3.0001 seconds (perhaps well within the acceptable range for a manual tester) will be marked as a failure.
If a test automation engineer removes all timing considerations, the tool might end up waiting indefinitely for an action to occur.

A test automation engineer needs to understand the requirements of their automation scripts and the way the tools operate. Selenium WebDriver with Python and Java, offers several wait mechanisms that can be used to synchronize automation execution. One of these mechanisms is explicit wait at the executing thread level, which should be used sparingly, yet is often one of the most commonly employed methods by many test automation engineers.

Explicit wait in Python at executing thread level

```
import time
# Pause execution for 5 seconds
time.sleep(5)
print("Resumed after 5 seconds")
```

Explicit wait in Java at executing thread level

```
public class SleepExample {
    public static void main(String[] args) {
        try {
            System.out.println("Pausing for 5 seconds...");
            Thread.sleep(5000); // Pause for 5000 milliseconds (5 seconds)
            System.out.println("Resumed after 5 seconds");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Although there are situations when such an approach may seem advantageous, it should be avoided in most cases. We have even seen extreme cases where test automation engineer become so dependent on it that the automation is executed at a speed slower than what a manual tester could do when performing the same test execution.

This wait mechanism is meant to cater for the worse case scenario. However, because such scenarios do not take place frequently, most of the waiting time is wasted unnecessarily. The sleep() function is helpful in the context of debugging test scripts.

The Selenium WebDriver offers two types of waiting - implicit waiting and explicit waiting.

An implicit wait in WebDriver is set when the WebDriver object is first created.

Implicit wait in Python at WebDriver level

```
from selenium import webdriver
# Initialize WebDriver (e.g., using Chrome)
driver = webdriver.Chrome()
# Set implicit wait time to 10 seconds
driver.implicitly_wait(10)
# Now any element lookup will wait up to 10 seconds before throwing an exception
driver.get('https://example.com')
# Example of finding an element (it will wait for the element to appear for up to 10s)
element = driver.find_element_by_id('example_id')
# Close the browser
driver.quit()
```

Implicit wait in Java at WebDriver level

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Example {
    public static void main(String[] args) {
        // Set the path for the WebDriver (e.g., ChromeDriver)
        System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
        // Initialize WebDriver (e.g., using Chrome)
        WebDriver driver = new ChromeDriver();
        // Set implicit wait time to 10 seconds
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        // Now any element lookup will wait up to 10 seconds before throwing an
exception
        driver.get("https://example.com");
        // Example of finding an element (it will wait for the element to appear for up
to 10s)
        WebElement element = driver.findElement(By.id("example_id"));
        // Close the browser
        driver.quit();
    }
}
```

The implicit wait specified remains until the WebDriver is killed. This wait setting is defined when the WebDriver is initialized, and it forces it in looking into the DOM for a given period when an

element is not found at once. Normally, the wait time preset is zero seconds. In case it takes too long to find the element by the code, WebDriver is set to search for it up to ten seconds more, which means asking over and over again every few seconds, Is the element here yet? If the element is found in that period, the script would continue its flow. The implicit wait applies to any element or elements referenced in the script.

However, explicit waits are such that the test automation engineer has to determine exactly how long WebDriver should wait for that particular element, mostly until that element has attained a certain state.

with the exception of the sleep() method, explicit waits are hassle free in their implementation because all Python, Java and C# bindings have nice methods for waiting for conditions that are expected. In Python and Java, these waits are coded by using the WebDriver method, ***WebDriverWait()***, in conjunction with an ExpectedCondition as discussed in section 3.3.

Example of ExpectedCondition in Python:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Initialize WebDriver (e.g., using Chrome)
driver = webdriver.Chrome()
# Navigate to the website
driver.get('https://example.com')
# Set explicit wait time (e.g., 10 seconds)
wait = WebDriverWait(driver, 10)
# Example: Wait for an element to be clickable before interacting with it
element = wait.until(EC.element_to_be_clickable((By.ID, 'example_id')))
# Interact with the element
element.click()
# Close the browser
driver.quit()
```

Example of ExpectedCondition in Java:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class Example {
    public static void main(String[] args) {
        // Set the path for the WebDriver (e.g., ChromeDriver)
        System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");

        // Initialize WebDriver (e.g., using Chrome)
        WebDriver driver = new ChromeDriver();

        // Navigate to the website
        driver.get("https://example.com");

        // Set explicit wait time (e.g., 10 seconds)
        WebDriverWait wait = new WebDriverWait(driver, 10);

        // Example: Wait for an element to be clickable before interacting with
it
        WebElement element =
wait.until(ExpectedConditions.elementToBeClickable(By.id("example_id")));

        // Interact with the element
        element.click();

        // Close the browser
        driver.quit();
    }
}
```

The list of ExpectedConditions in Python and Java is given in section 3.3.

Some browser drivers also allow some interface to check the page loading state. For example, when in Google Chrome driver, a website is being loaded, there is loader icon animation on the Google Chrome tab. This page load status can thus be check for test automation purpose. One

way of doing this is by polling the page status using a JavaScript hook in the browser. The JavaScript hook returns a status “COMPLETE” only when the page is fully loaded. The polling can thus be used as a waiting mechanism as shown below.

Example of polling of page status in Python:

```
# Wait for the page to load completely
while_limit = 0
while driver.execute_script("return document.readyState") != "complete" and
while_limit < 180:
    time.sleep(1)
    while_limit += 1
```

Example of polling of page status in Java:

```
int whileLimit =0;
String WDState = ((JavascriptExecutor) driver).executeScript("return
document.readyState;").toString();
while (!WDState.toUpperCase().equals("COMPLETE") && whileLimit<180)
{
       sleep(1000);
    whileLimit++;
}
```

4.3 Page Objects

A Page Object represents an area in the web application interface with which the test is going to interact. Below are some reasons to use page objects in automated tests:

- They create reusable code that can be shared by multiple test scripts.
- The reduce the amount of duplicated code.
- They reduce cost and maintenance efforts for test automation scripts.
- They encapsulate all operations on the SUT’s GUI in one layer.
- They give a clear separation between the business and the technical parts for the test automation design.
- When change inevitably occurs, it gives a single point to repair all relevant scripts.

Page Object Pattern means using Page Objects in the Test Automation Architecture. One of the core principles of building a maintainable TAA is dividing it into layers. One of the layers of generic

TAA (as defined in the ISTQB® TAE v2.0 syllabus) is the Test Definition Layer; this layer takes care of interfacing between the logic of the test case and the physical needs of driving the SUT. Page objects are part of this layer, usually abstracting the GUI of the SUT. Abstracting means to hide details of how we are controlling the GUI inside functions that are used in other scripts.

Below is a fragment of code that might appear in a script without using the Page Object Pattern. Notice that it is not easily readable; the locators for specific controls are tied intimately with the code.

```
def test_valid_login(driver):
    # Directly interacting with the elements on the login page
    username_input = driver.find_element(By.ID, 'username')
    password_input = driver.find_element(By.ID, 'password')
    login_button = driver.find_element(By.ID, 'login-button')
    # Performing actions on the elements
    username_input.send_keys("valid_user")
    password_input.send_keys("valid_password")
    login_button.click()
```

After applying page pattern, the same code fragment looks as below:

This class will represent the login page and include methods for interacting with elements on that page.

```
from selenium.webdriver.common.by import By
class LoginPage:
    def __init__(self, driver):
        self.driver = driver
        self.username_input = (By.ID, 'username')
        self.password_input = (By.ID, 'password')
        self.login_button = (By.ID, 'login-button')
        self.error_message = (By.ID, 'error-message')
    def enter_username(self, username):
        self.driver.find_element(*self.username_input).send_keys(username)
    def enter_password(self, password):
        self.driver.find_element(*self.password_input).send_keys(password)
    def click_login(self):
        self.driver.find_element(*self.login_button).click()
```

This is where the test logic will reside. It uses the LoginPage class to perform actions on the login page.

```
import pytest
from selenium import webdriver
from login_page import LoginPage

def test_valid_login(driver):
    login_page = LoginPage(driver)
    login_page.enter_username("valid_user")
    login_page.enter_password("valid_password")
    login_page.click_login()
```

The codes are now more readable compared to the format which did not use page pattern.

We have effectively abstracted much of the script's inherent complexity and transferred it to the Test Automation Architecture (TAA). It is crucial to understand that the complexity remains; it is the very element that enables us to develop a robust automation script, and its presence is necessary. However, by concealing this complexity, we simplify the process for script developers, allowing them to create valid and powerful scripts more efficiently.

This approach of managing and concealing complexity is recognized as a best practice in programming. Page Object is a class, module, or set of functions that encapsulates the interface to a form, page, or fragment of a page within the System Under Test (SUT), which a test automation engineer aims to control.

A Page Object provides business operations that are utilized as test steps within the Test Execution Layer. The Page Object Pattern helps to reduce the maintenance effort within a project as it can substantially decrease the time required to update scripts following changes to the SUT's graphical user interface (GUI).

Be aware that the Page Object Pattern is not a cure-all for challenges associated with test automation maintenance. While it can reduce costs, there are instances—such as when the logic of the SUT changes—where updating test scripts can still be time-consuming. This issue is not unique to automation testing; if the SUT's logic changes, manual test cases will also require updates. Fortunately, changes to SUT logic are infrequent, as they would necessitate users relearning how to operate the application.
When structuring the Test Automation Architecture (TAA) into layers and designing Page Objects, the below key principles should be adhered to:

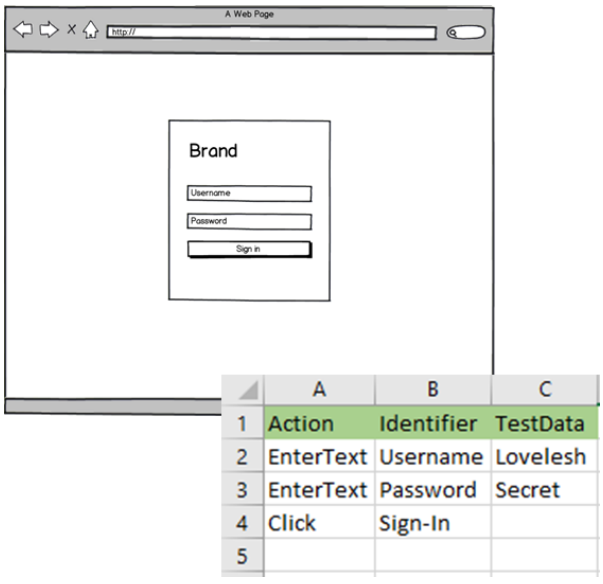
- Page Objects should not contain business assertions or verification points.
- All assertions and technical verifications related to the GUI (e.g., verifying that a page has fully loaded) should be handled exclusively within Page Objects.
- All waits should be encapsulated within Page Objects.
- Only the Page Object should make calls to Selenium functions.
- A Page Object does not need to cover the entire page or form; it can control a specific section or part of it.

In a well-designed TAA, a Page Object within the Test Definition Layer encapsulates calls to Selenium WebDriver methods, ensuring that the Test Execution Layer interacts only with the business logic as such there is no need for importing or using the Selenium library within the Test Execution Layer.

4.4 Keyword Driven TAS

A keyword driven automation solution is a TAS design pattern that focusses on code re-use and ease of test maintenance. The test data and the test cases are kept external to the TAS, which is not in the source code of the TAS. The test case may include the object identifiers, the actions or interactions that need to be carried out and the test data needed to carry out the test.

To explain keyword driven automation design, consider the following example:



	A	B	C
1	Action	Identifier	TestData
2	EnterText	Username	Lovelesh
3	EnterText	Password	Secret
4	Click	Sign-In	
5			

```
public class ExecuteTest
{
	public void PerformAction( identifier,action,testdata)
	{
		switch (action)
		{
		case (EnterText)
		{
		driver.findElement(By.name(identifier)).sendKeys(testdata);
		}
		case (Click)
		{
		driver.findElement(By.name(identifier)).click();
		}
	}
}
```

In this example, the external file used is a spreadsheet storing the test case and the test data. To login, we need to enter the username, enter the password, and click on “Sign-In” button. Therefore, the action to enter a text in a textbox is to be repeated twice. Here, the action to enter a text is coded only once. There is a selection construct to invoke the needed interaction. In the example, it is structured as a case construct based on the action that is scripted in the externalized file. In this case, the actions used are called unit keywords because they cannot be broken down into other simpler actions. For example, we cannot break the action click into simpler interactions with the system.

Alternatively, we can combine multiple unit actions or keywords to form compound keywords. Compound keywords represent high-level business interactions with a system that are meaningful for the test analyst. In this example, the low-level keywords of ‘EnterText’ and ‘Click’ can be combined to create the high-level keyword ‘Login’. The ‘Login’ action will encompass the action of entering the username, the password and clicking on the “Sign-in” button. This has the advantage of avoiding repetitive test steps and thus making the test case more compact, but it also takes away some flexibility from the test analyst in case the login process changes. If there is a change in the login process, then the compound action ‘Login’ will need to be maintained in the source code.

Hence, the key in this framework design is finding the right balance between the needed level of abstraction to make test scripting easy and keeping the maintenance flexibility. This TAS design derives its name from the fact that the “action” keyword drives the test execution.

Keywords have several places of definition and usage in the generic Test Automation Architecture:

- Low-level keywords are implemented as Page Objects in the Test Adaptation Layer, they do the actions on the SUT
- Low-level keywords are used as parts of higher-level keywords in the test library of the Test Definition Layer
- High-level keywords are implemented in the test library

High-level keywords are used as test steps of test procedures in the Test Execution Layer.

This approach offers several benefits:

- **Decoupled test case design:** The logic of test cases decoupled from the TAS
- **Clear layer separation:** There is a distinct separation between the Test Execution, Test Abstraction, and Test Definition layers.
- **Division of labor:**
    - Test analysts design test cases and create scripts using keywords, data, and expected results.
    - Technical test analysts / automation engineers implement the keywords, and the execution framework required to run the tests.
- **Keyword reusability:** Keywords can be reused across different test cases.
- **Enhanced readability:** Test cases become more readable and resemble manual test cases.
- **Reduced redundancy:** There is less duplication of effort.
- **Lower maintenance costs:** Maintenance becomes less costly and time-consuming.
- **Manual execution option:** If test automation is unavailable for some reason, a manual tester can execute the test manually directly from the automated script.
- **Early script development:** Since keywords are abstract, scripts using keywords can be written before the SUT is available for testing, similar to manual tests.
- **Early automation:** Test automation can be prepared earlier, allowing it to be used in functional testing, not just regression testing.
- **Scalability:** A few test automation engineers can support many test analysts, making the architecture highly scalable.

- **Tool flexibility:** As the scripts are completely separated from the implementation level, different tools can be used interchangeably.
- **TAS size:** The size of the TAS is solely dependent on the number of keywords we have in the framework. As such, when the number of test cases grows, the TAS size remains the same and only the external file size grows.
- **Compilation of the TAS:** Maintenance of keywords only requires recompilation of the TAS. Thus, if test cases are changed, the TAS does not require re-compilation.

This framework does nevertheless have some drawbacks. Given that the underlying architecture is more complex than a page pattern framework, the coding of this framework is more complex and therefore requires more time. The externalized file designs and streams need to be coded correctly so that test date and test scripts are maintainable in the long term. If these designs are not flexible and not extensible, then the framework may need to be coded repeatedly for each small change in the design. For this reason, the framework design requires careful design upfront adapted to the business need. This pre-coding task defines the success of the framework in the organization. It is also undeniable that those doing the test scripting do need some basic training on how to extract the object identifiers from the SUT in a standard way.

Version control is of the utmost importance in keyword driven automation framework, as the externalized file and the framework need to work hand in hand. If incorrectly version controlled, test cases, test data, and the test automation solution source code changes can be lost or overwritten.

A keyword-driven test automation framework does require foresight from the test automation engineers as its success is heavily dependent on the flexibility and extensibility of the design. Therefore, when implemented correctly, this framework design brings a quick return on investment.

When implementing a TAS based on keyword-driven principles, the following aspects should be considered:

- Granular keywords allow more specific scenarios, but at the cost of script maintenance complexity.
- The more fine-grained a keyword is, the more likely it is tied intimately to the interface of the SUT and the less abstract it is (e.g., the example, “Click >>Cancel<< button”.
- Initial keyword design is important, but new and different keywords will ultimately be needed, which implicates both business logic and the automation functionality to execute it

When implemented correctly, a keyword-driven automation solution can be an effective approach that leads to scripts with consistently low maintenance costs. It enables the creation of well-structured test automation frameworks and incorporates best practices in software design. SO/IEC/IEEE 29119-5 is a standard that provides guidelines for keyword-driven testing in software testing. The standard focuses specifically on the creation, management, and execution of tests using a keyword-driven approach, enabling the reuse of test scripts and improving automation efficiency.

5 Maximizing ROI on Test Automation – 150 minutes

Learning Objectives

STF5-1 (K2) Understand the advantages and disadvantages of headless automation testing

STF5-2 (K2) Understand how machine learning can help in reducing false positives and maintenance effort

STF5-3 (K2) Understand test parallelism and its advantages

5.1 Headless Test Automation

Selenium WebDriver's UI automation relies heavily on web browsers. An essential component of web test automation is opening a browser and running the test cases on it. However, we frequently hit problems when running Selenium tests on any of the browsers, such as poor browser rendering, conflicts with other software programs, etc. Additionally, many CI/CD systems are non-UI (such as Unix based systems). This may hinder running automated tests on UI browsers in the CI/CD pipeline.

As a result, to run the test cases on those systems, there needs to be a method to run the tests without the UI. This is where headless browser testing becomes essential. It facilitates the execution of Selenium Headless Browser tests without displaying any user interface. The headless browser can examine and comprehend webpages directly in the memory. As a result, a headless browser can give the real browser perspective without heavily consuming the system's resources (memory and CPU time). The time it takes a typical browser to load CSS, JavaScript, and render HTML is saved because the browser GUI is not opened. Ten times quicker performance scaling can be noted with headless browser testing. The headless browser might be a favorable option if performance and system resource consumption are crucial factors.

Selenium uses HtmlUnitDriver, a different WebDriver implementation that is comparable to FirefoxDriver, ChromeDriver, and other WebDriver implementations, to provide headless testing. The library must be manually added to use HTMLUnitDriver, which is available as an external dependency. HTMLUnitDriver is meant to be used within another testing framework, such as Junit or testing, and is not a general-purpose unit testing framework. It is a method for testing purposes to simulate a genuine browser like Chrome, Safari, or Firefox.

The below code snippets show the implementation of a headless browser on Chrome in Python and Java.

Defining a headless Chrome browser in Python

```
# Set up Chrome options
chrome_options = Options()
chrome_options.add_argument("--headless")
# Initialize the Chrome WebDriver
driver = webdriver.Chrome(options=chrome_options)
```

Defining a headless Chrome browser in Java

```
// Create a new instance of the ChromeDriver
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless");
WebDriver driver = new ChromeDriver(options);
```

The benefits of running a Selenium test case in headless mode include:

- Faster execution of automated tests
- Multi-tasking
- Useful for web scraping
- Multiple browsers support
- Possibility of executing tests on non-UI operating systems.
- Easier integration in the CI (continuous integration) workflow

Although headless browsers have many benefits, there are a few considerations to keep in mind when using them:

- Running tests in headless mode may make the debugging process more challenging as no GUI can be used to identify failure points.
- Due to the pace at which tests are executed, headless browsers do not accurately simulate user activity, and therefore test results may include false negative.
- Exploratory testing, visual regression testing, and other forms are testing are restricted due to the absence of UI.

5.2 Parallelism of tests

The ability to run tests in parallel comes with a lot of advantages. The main one is that the test execution time is reduced significantly, and the test results are available even earlier. Given that tests are executed in parallel, it also implies that more tests can be conducted in the same time lapse.

There are two concepts when doing parallel tests. Separate test suites can be executed on different browsers concurrently on the same machines or separate test suites can be executed on different machines concurrently.

5.2.1 Selenium automation and performance testing

It is advantageous to use Selenium automation tools for performance testing. The tests are indeed scalable, but care should be taken to limit the probe effect.

The probe effect is defined as the effect on the component or system by the measurement instrument when the component or system is being measured. Performance testing is about the measurement of metrics such as response time, usage of processing capacity and memory, etc. These measurements can be tampered with if the TAS is not properly designed and built.

The probe effect is most prevalent when a single machine is used to simulate multiple users. This will be implemented as multiple execution threads on a single machine and may entail that threads are blocked or enter a deadlock state. Each thread will instantiate a browser controller and the browser to allow execution. Therefore, some lags are expected in response time due to the overload in processing and memory usage. For example, if the TAS is used to simulate 20 users on a single machine using Google Chrome browser, it will mean that 20 Chrome browsers will initialize, and the actions will be done concurrently. The system will start lagging and the response time for the actions and tests will be disturbed. The device lag will introduce a probe effect in the test results.

Therefore, this should be considered when using the TAS for performance testing. Ideally, the test architecture should enable the execution of tests on multiple devices concurrently and in headless mode to limit the probe effect to its minimum. The usage of Selenium Grid is very convenient then for these purposes.

5.3 Machine learning and test automation

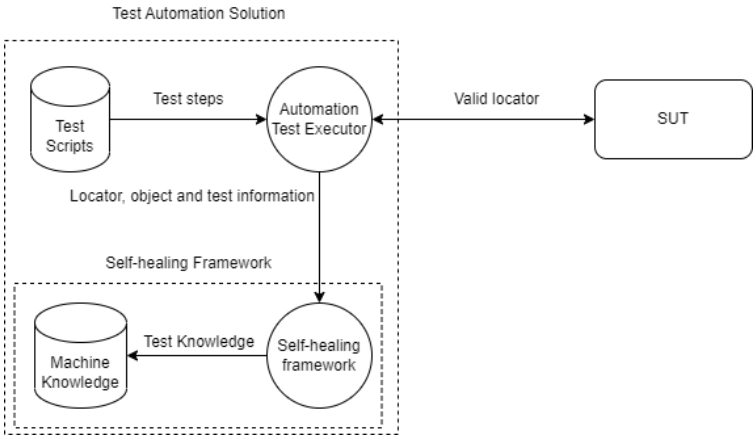
Using artificial intelligence in the field of automation is relatively new. The idea is to have the capability to rely on the machine's ability to learn how to recover from failure or false positives to keep the test going.
Machine learning is a branch of artificial intelligence that focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving the AI robot accuracy.

5.3.1 Self-healing tests

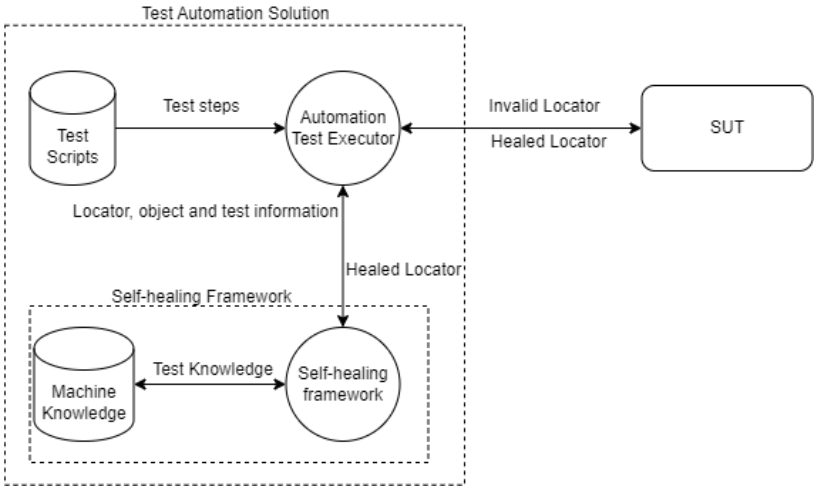
According to a study made by ResearchGate (Efficient and Change-Resilient Test Automation: An Industrial Case Study Authors: Sebastian Bauersfeld, Martin Fewster, and Andreas Zeller) 74% of tests are intermittently failing due to brittle locators. In other words, the locators used in the tests are not good or strong enough to sustain the tests over long periods of test execution. This may happen due to changing the SUT, the technology used, and the way the tests are written. In any case, this implies that tests are required to be continuously maintained, and this does hinder the increase in test coverage and reduce the return on investment of test automation initiatives.

Self-healing testing is an approach of using machine learning to allow broken locators to be repaired to keep the tests operational and help in test automation maintenance. The way the mechanism works is that the self-healing framework plugs into the TAS and monitors on-going tests. The self-healing framework learns about the tests, locators and objects being interacted with when the tests are running.
In another test run, assuming that there was a slight change in the SUT, and some locators are no longer valid, when the invalid locators are attempted, the self-healing framework takes over and suggests alternative locators (healed locators) that may work for the test. This way, the tests continue and still work without immediate test maintenance. Maintenance will still be necessary, but it can be done at a later stage. The operation of the self-healing framework is summarized in the diagram below.

Learning phase:



Healing phase:



There are both commercial and open-source self-healing frameworks. One self-healing framework that is made for Selenium-based project is Healenium. Healenium provides the TAE with a detailed report at the end of the tests to describe which locator was healed. Healenium integrates well with Java-based Selenium projects and supports concurrent test execution.

5.4 Best practices

To maintain the success of a test automation project, it is important to have some best practices on the underlying process of test scripting and TAS coding. The test automation project is a development project which implies that the best practices for coding are expected. It is expected that the TAS code is well annotated, version controlled, and indented, the source code is modularized to facilitate technical maintenance and the coding standard to denote the nomenclature for variable names, database fields, etc. is maintained.

5.4.1 Test Scripts Acceptance Criteria

When scripting an automated test case, it is recommended to benchmark the test case against a certain quality standard, also known as a quality gate or acceptance criteria. This ensures that test scripts are checked for completeness and the strength of locators used.

For example, in a login test case, it is beneficial to add multiple verification points to detect possible defects. Some verification points may include:

1) Verify that the username is displayed on the page.
2) Verify that the password is input masked and is displayed as an asterisk.
3) Verify that the login button is enabled when username and password are typed in.
4) Verify that the URL changes (or not) after login is attempted.
5) Verify that an entry / update is made in the database if the login is successful and if the TAS has an interface with the database.

These verification points make the test more comprehensive. While these steps may be time-consuming to perform manually each time the login test is executed, they are quick to execute once scripted for test automation.
It is also very important to ensure that the locators used are strong. One way to ensure this is manual verification of the locator. For example, ensuring relative XPaths are used or ensuring unnecessary attributes are not referenced in a locator.

Another way is to run the test multiple times over different test environments to ensure that the locators and the test case logic are not environment dependent.
For example, one test case can be executed 3 times in a test environment and 3 times in a pre-production environment. The acceptance criteria for the test case can then be that all 6 test runs are successfully passing. This verification is quite useful for detecting brittle locators and therefore avoids costly false positives.

5.4.2 Choice of selectors and locators

The choice of the locator is important in the test automation.

The criteria for a good locator are as below:

1) A locator (such as XPath) that stays constant. This will minimize test maintenance. Some development technologies change the DOM elements with each compile or build, meaning the IDs of elements may not remain constant. Using such selectors and locators would be ineffective.

2) A locator that can be used to uniquely identify web elements. Given that most common actions on a Selenium automation tool require a web element to be uniquely identified, it makes sense to use attributes that make the element easy to uniquely locate. At times it may be required to use conjunction operators such as the ‘and’ / ‘or’ keyword to be able to uniquely identify a web element.

3) A locator that has meaning.
   Having a meaningful value in the selector and locator is always helpful when following the test steps whether for review or maintenance. This will cut down the need to open the SUT and manually check which element is referred to in the test.

   For example, the below locator (.//*[contains(@type,"email")]) is more intuitive than (.//*[contains(@id,"text1091")]).

It is always a good idea to work closely with the development team when building the TAS and writing automated test scripts. Sometimes, the development team of the SUT can inject a specific attribute into the DOM element that could be used for test automation.

6 Bibliography

ID	Author	Resource / Title	Edition / Version
01	ISTQB®	Test Automation Engineering Syllabus	2024
02	ISTQB®	Test Automation Strategy Syllabus	2024
03	ISTQB®	Certified Tester Foundation Level Syllabus	2024
04	ISTQB®	Standard Glossary of Terms used in Software Testing Version 4.0	2018
05	Telerik Test Studio	10 tips on how to dramatically reduce test maintenance	2017
06	Selenium Project	https://www.selenium.dev/	2022
07	Larry Yang	Factors to Consider When Implementing Automated Software Testing	2016
08	Software Testing Magazine	Self-Healing Automated Tests	2022
09	Pratham Software (PSI)	Common Selenium Exceptions	2021
10	Aditya Garg and Pallavi Sharma	Selenium 4 - A quick and practical guide	2021
11	Baiju Muthukadan	Selenium Python Bindings	2022
12	Tutorials Point	XPath query language for XML	2018
13	Tutorials Point	Cascading Style Sheets	2017

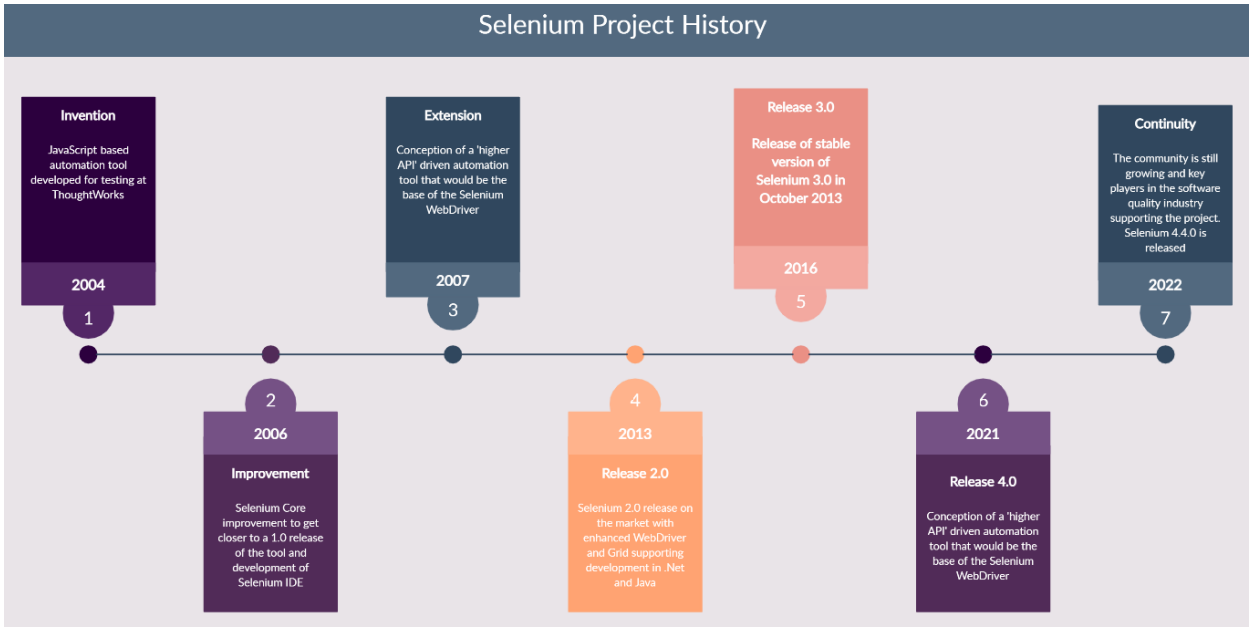
7 Appendix A

7.1 History and evolution of the Selenium automation tool suite

Selenium test automation tools have been around in the software quality area since 2004 and since then the tools have gained increasing traction and support from the community. The Selenium project started in Chicago at Thought Works (by Jason Huggins) to test an internal application in the company.

The project became popular over the years and even today the Selenium community and contributors continue to grow. The project is supported by major organizations in the IT industry.

The diagram below summarizes key events in the Selenium project.



7.2 Learning objectives/cognitive level of knowledge

The following learning objectives are defined as applying to this syllabus. Each topic in the syllabus will be examined according to the learning objective for it. The level of Knowledge is the same as per the International Software Testing Qualifications Board (ISTQB®).

Level 1: Remember (K1)

The candidate will recognize, remember, and recall a term or concept.

Keywords: Identify, remember, retrieve, recall, recognize, know

Level 2: Understand (K2)

The candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify, categorize, and give examples for the testing concept.

Keywords: Summarize, generalize, abstract, classify, compare, map, contrast, exemplify, interpret, translate, represent, infer, conclude, categorize, construct models

Level 3: Apply (K3)

The candidate can select the correct application of a concept or technique and apply it to a given context.

Keywords: Implement, execute, use, follow a procedure, apply a procedure

Level 4: Analyze (K4)

The candidate can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences. A typical application is to analyze a document, software or project situation and propose appropriate actions to solve a problem or task.

Keywords: Analyze, organize, find coherence, integrate, outline, parse, structure, attribute, deconstruct, differentiate, discriminate, distinguish, focus, select