Saint Louis University
SCHOOL OF ACCOUNTANCY, MANAGEMENT, COMPUTING and INFORMATION STUDIES
Department of Information Technology

Software Engineering Fundamentals

Project No. 1 – Software Testing

Selenium

List of Members:

ALINGAY, Jillah Marie

BALDERAS, Peter Bruce

DIMACALI, Paul

DE GUZMAN, Crisha Mae

MACATUGGAL, Julliard

MADISON, Jhoie Amber

MAYANGAO, Mike

SANTOS, Rhem Mc Garth

Class Code:

9482

Submission Date:

Dec 9, 2023

Table of Contents

PART 1: Testing Concepts

1.1 The objectives of software testing

Software testing is an important aspect of the software development lifecycle which is aimed at finding errors in a program or assessing any activity related to program's functionality. It plays a necessary role in achieving several objectives throughout the software development lifecycle. Below are the objectives of software testing:(Lazic, L 2010)

1. **To Improve Quality:**
    - Software testing is essential in enhancing the quality and reliability of software that is being developed.
    - Software bugs can have severe consequences which can lead to failures and disasters, making the quality and reliability of software a big concern.
    - In software development, quality is defined as conformance to specified design requirements, and testing is an important method to identify and eliminate design defects.
    - Debugging is heavily performed to identify and correct the identified defects, ensuring that the software performs as required under specified circumstances.
2. **For Verification and Validation (V&V):**
    - Testing constitutes a part of the verification and validation activity.
    - It assesses if a product works or has issues.
    - Although whole system direct testing may not always be possible, factors related to testing serve for the verification and validation of the software.
3. **For Reliability Estimation:**
    - Software testing is more of an art than science; it involves using creative methods and practical strategies.
    - Testing techniques used today may not resemble those from 30 years ago, emphasizing the evolving nature of testing.

- Despite advancements, software testing is costly, especially when software's quality is at stake, and absolute certainty regarding the correctness of specifications and software cannot be guaranteed.

In this context, Selenium is a powerful tool that aligns with the objectives of software testing by providing a comprehensive set of features (Selenium IDE. nd)

1. **Language Support:**
    - Selenium supports multiple programming languages, enabling testers to choose the language that best suits their needs enhancing flexibility in the creation of test cases.
1. **Browser Support:**
    - Selenium allows testing across various browsers, ensuring compatibility and functionality across different platforms contributing to the verification and validation for cross-browser compatibility.
2. **Scalability:**
    - Selenium's scalability allows it to cover many test cases and scenarios, supporting improving software quality. Scalability, in a sense, where different frameworks can be integrated.
3. **Reusable Test Scripts:**
    - Selenium enables the creation of reusable test scripts, promoting efficiency in test case creation and maintenance, which promotes software quality.
4. **Parallel Testing:**
    - Selenium's support for parallel test execution reduces overall testing time, contributing to faster and more efficient development process.
5. **Documentation and Reporting:**
    - Selenium provides a test execution logs and reports, identifying issues and supporting the software testing process' goal..
6. **User Experience Testing:**

- Selenium's ability to simulate user interactions contributes to assessing and ensuring a positive user experience, aligning with the objective of software quality improvement.

7. **Continuous Integration and Continuous Deployment (CI/CD):**
   - Selenium's integration into CI/CD pipelines automate testing, facilitating early issue identification and contributing to faster and more reliable releases, supporting the overall software testing process.

1.2 Software quality and project trade-offs

Software quality is the satisfaction of customer needs and the absence of deficiencies. It encompasses planning to identify requirements, control through evaluation against customer specifications, and continuous improvement to maintain quality. This involves resource allocation, training, and establishing a permanent structure for ongoing quality pursuits in software development.(Carty D., 2018)

Addressing tradeoffs in software development is vital for achieving high-quality outcomes. Choices like multithreading impact responsiveness but introduce complexity. The dynamic versus static allocation decision balances ease of use in object-oriented programming against the overhead of expensive memory operations. Trading space for time involves optimizing between memory and CPU processing efficiency. The tension between performance and flexibility emerges in decisions like compile-time versus runtime linking of object files. Furthermore, conflicts may arise when prioritizing ease of use, as with user-friendly wizards, potentially sacrificing performance. Successfully navigating these tradeoffs is central to delivering software that not only meets functional requirements but also excels in performance, resource utilization, and user experience.(Kumares S., et al, 2010)

1.3 Defects: faults and failures

In software testing, a defect signifies a deviation from user or business requirements, representing coding issues that can impact the entire program. These flaws, introduced during the development cycle, result from mistakes made by programmers. Common defect types include:

- Arithmetic defects stemming from errors in mathematical expressions.
- Syntax defects arise from coding mistakes.
- Logical flaws, rooted in misunderstandings or oversight of requirements.
- Performance defects occur when the software fails to meet expected results.
- Multithreading defects arise during simultaneous task execution and potentially lead to system failure.
- Interface defects hindering user-software interaction.

Though bugs and defects share similarities, both demand immediate correction before deployment. Addressing these diverse defect types is crucial for ensuring software efficiency, functionality, and user satisfaction (Baeldung 2023).

Software failure takes place due to errors resulting from a breakdown of an application in the process of its operation. Defects do not always result in losses but they could be caused by unfavorable conditions of the environment, human errors and acts of fraud intentionally. Post test analyzes are carried out in order to ascertain why incidents occurred during testing and thus the importance of pre-deployment quality checks aimed at improving consumer confidence. Comprehensive testing is a key factor in building software robustness and reliability, for defective parts only contribute to failures as they execute (Baeldung 2023).

Software faults are deviations of expected behavior resulting in warning notices but failure without correction. They can be categorized into the following groups: algorithm faults, syntax errors, computational problems, timing errors, document inconsistencies, overload cases, hardware-software incompatibility, omission or commissions mistakes. Among the preventive measures are programming methods, techniques, peer review, code analysis as well as

robust requirements in support of software and hardware to guarantee quality product. Appointing appropriate methods while proactive in order to prevent faults in order to create an overall reliable program (Baeldung, 2023).

1.4 The cost of fixing a defect

Similar to the preventive measures implemented in the medical field, the saying "prevention is better than cure" appropriately applies to defects in the software development life cycle. The costs of locating and fixing bugs increase as a program advances. Bug fix post release could be expensive too and risky that may require further alteration which will consume much more of money, time, and labor.(Testing Guru, 2020)

The Systems Sciences Institute at IBM underscores the financial impact, estimating that correcting a glitch discovered during implementation is six times costlier than identifying it during development. Fixing errors post product launch costs four to five times more than those discovered during design and up to 100 times more than issues identified during the upkeep phase. This cost escalation underscores the importance of early defect detection in the software development life cycle, with timely identification and correction being crucial for an efficient and cost-effective software quality assurance process. (Testing Guru, 2020)

PART 2: Testing Process

2.1 Software testing life cycle

In the context of Selenium, software testing life cycle includes planning, designing, executing, and managing the testing process using tools. It begins with the identification of requirements and tests of execution, defect tracking, and reporting. Selenium's integration testing life cycle facilitates the automation of functional and regression tests, to ensure thorough coverage and efficient testing processes.

The Types of Testing According to (Selenium, n.d):

1.) **Acceptance Testing:** Selenium conducts acceptance testing to validate if the system has met the specified acceptance criteria and if it's ready for deployment. This is to check whether the system satisfies business requirements and functions that the user has expected.
2.) **Functional Testing:** Selenium is used for functional testing to make sure that the web applications are functioning as expected. The testing involves the application's features and functions to ensure the requirements are working.
3.) **Integration Tests**: Selenium can be used for integration testing to validate the components are working as they are expected to. This ensures that the integrated parts of the application are functioning as a whole.
4.) **System Tests:** Selenium tests the entire software system as a whole to make sure that the components are working together seamlessly. This is to validate that the integrated system meets the requirements and functions correctly according to the target environment.
5.) **Performance Testing**: Selenium tests the systems responsiveness, speed, and performance with different conditions. This is to make sure that the application can perform well with normal and peak load conditions.
6.) **Load Testing**: Selenium can be a part of testing scenarios where virtual users can interact with the application simultaneously. This is to assess how the system will perform during expected, peak, and stress conditions.
7.) **Stress Testing**: Selenium stress testing is done to verify the application when it comes to being under stress. This is to know where the system's breaking point is and to understand its resilience under different conditions.
8.) **Regression Testing**: Selenium is well-suited for regression testing, where the automated test scripts are used to confirm that the new code changes do not affect the existing functionality. This helps in catching unintended side effects of modifications in the code.

2.2 Defect injections and defect causes

Selenium has an important role in identifying defects by the automation of the testing process. Defect injections, or issues that have been discovered during the development, can be easily seen early through the automation of testing

scripts of Selenium and because of this, this reduces the chances of defects reaching production. The common defect causes such as coding errors or compatibility issues, can be addressed through Selenium's testing capabilities, and that will enhance the quality of the software. (Selenium, n.d)

According to (Selenium, n.d) this are the primary defects and its causes

1.) **Coding Errors**: Defects that can be found are the mistakes when coding scripts or the application itself. Typos, logical errors, or the wrong syntax in the script may be the cause of coding issues.
2.) **Compatibility Issues**: Since Selenium is used for web browsers compatibility may be found when the application is tested on different browsers with different versions.
3.) **Element Identification**: Selenium relies on the identification of web elements to interact with the application.
4.) **Timing Issue**: Scripts in Selenium may encounter defects if the timing is not right. An example can be where a delay in the page is loading. When timing is not right problems can lead to script failures.
5.) **Environmental Factor**: Defects can also be found here. The incorrect configuration, settings, or dependencies between the development and testing environments can give unexpected results or issues.

2.3 Testing philosophies

Selenium has a lot of testing philosophies, and that includes agile and continuous testing. The flexibility of this integration allows development cycles, supporting rapid, and iterative testing. Selenium's compatibility with the continuous integration of tools enables the implementation of continuous testing philosophies, and that ensures that testing is an essential part of the development pipeline, having faster feedback and early defect resolution. (Selenium, n.d)

2.4 Testing pipeline

Selenium has an easy integration of the testing pipeline, automatic repetitive testing tasks, and speeding up the delivery process. It can be included into continuous integration/continuous deployment (CI/CD) pipelines, and this makes sure that the test suites are executed automatically as a part of the software delivery process. This integration enhances the reliability of the testing process, promoting an efficient and streamlined pipeline. (Selenium, n.d)

Implementing a test pipeline for Selenium:

1.) **Choose a CI/CD tool**: Select a CI/CD tool that will fit our project. The popular choices are Jenkins, GitLab CI/CD.
2.) **Setting Up Selenium WebDriver**: Making sure that the webdriver is in the testing environment.
3.) **Version Control System**: Ensure that the codebase is managed by a control system like Git.
4.) **Write Selenium Test Scripts**: Create a Selenium test script with Java, Python, or any preferred programming language.
5.) **Create a Test Suite**: This allows it to run multiple tests in an order to manage them.
6.) **Configure CI/CD Pipeline:** Set up a configuration file for the chosen CI/CD tool. To test any pre-test or post-test action.
7.) **Install Dependencies:** Make sure that Selenium WebDriver is installed on the CI/CD server
8.) **Trigger Pipeline on Code Changes:** Configure the CI/CD pipeline to automatically trigger whenever there are changes to the code repository.

PART 3: Defect Detection Approaches

3.1 Test coverage and testing dimensions

Selenium is a popular open-source automation framework for web applications (Selenium, n.d.). It allows developers to create automated tests for various functionalities and user interactions. However, detecting defects with Selenium alone requires a comprehensive approach that considers both test coverage and various testing dimensions (Hamilton, 2023).

Test Coverage

Test coverage refers to the extent to which your test cases cover different functionalities and scenarios of your web application. Achieving high test coverage is crucial for ensuring that most potential defects are identified during the testing process (Nair, 2022). These are some key aspects of test coverage in the context of Selenium:

- *Code Coverage*: Measures the percentage of code exercised by test cases. This helps identify untested functionalities and areas potentially harboring defects (Toledo, 2018).
- *Functionality Coverage*: Ensures that the tests cover all key features and functionalities of the application (*Types of Testing*, 2023).
- *User Journey Coverage*: Represents the extent to which tests cover typical user journeys through the application. This helps identify defects that could impact user experience (Gerrard, Collier, & Huggins, 2021).
- *Data Coverage*: Tests should use different sets of data to ensure functionality is consistent across various scenarios (Ashawami, 2022).

Testing Dimensions

While achieving good test coverage is essential, it's equally important to consider various testing dimensions for effective defect detection. These dimensions help identify defects that might be missed by focusing solely on code coverage (*How to Get Height and Width of Element in Selenium WebDriver*, n.d.). These are some key testing dimensions:

- *Usability Testing*: Evaluates how easy and intuitive the application is to use. This can help identify defects that impact user experience and satisfaction (Vijay, 2023).

- *Accessibility Testing*: Ensures the application is accessible to users with disabilities. This includes checking for compliance with accessibility guidelines and testing the application with assistive technologies (*Accessibility Testing With Selenium Webdriver With Code Example*, 2023).
- *Compatibility Testing*: Tests the application's compatibility across different browsers, operating systems, and devices. This helps prevent defects that arise from compatibility issues (Hamilton, 2023).
- *Performance Testing*: Evaluates the application's performance under load. This helps identify defects that impact responsiveness and scalability (Baskirt, 2022).
- *Security Testing*: Identifies vulnerabilities and security weaknesses in the application. This helps prevent security breaches and data leaks (*Security Testing and Selenium*, 2021).

3.2 White-box versus Black-box testing versus Gray-box testing

Black-box testing

Selenium, a versatile automation tool, finds its application across various testing methodologies. Its ability to simulate user interactions and verify application behavior makes it valuable for black-box testing . Testers leverage Selenium to write test cases that mimic real user journeys, uncovering functional errors, usability issues, and performance bottlenecks . This black-box approach ensures the application functions as intended from the user's perspective (Selenium, n.d.).

Gray-box testing

Selenium utilizes partial knowledge of the internal workings. Testers can access specific elements and data structures, allowing them to write targeted test cases for integration issues, security vulnerabilities, and compatibility problems. This approach bridges the gap between black-box and white-box testing, offering a more comprehensive perspective (Selenium, n.d.).

White-box testing

Selenium shines by directly interacting with the application's code. Testers can write test cases that execute specific code paths, verifying logic errors, data flow issues, and code coverage. This granular approach provides valuable insights into the application's inner workings, ensuring efficient and comprehensive testing (Selenium, n.d.).

Selenium's adaptability allows it to seamlessly integrate with different testing approaches, making it a valuable tool for testers of all levels. From black-box's user-centric focus to white-box's deep code analysis, Selenium empowers testers to ensure the quality and reliability of software applications (Selenium, n.d.).

3.3 Scripted versus non-scripted tests

Scripted Testing

Scripting involves writing code in a programming language like Python or Java to automate test cases (Crispin & Gregory, 2009). These scripts control the browser, perform actions like clicking buttons, filling forms, and verifying outputs (Gundecha, 2015).

```java
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class SeleniumExample {

    public static void main(String[] args) {
        System.setProperty("webdriver.gecko.driver", "D:/Downloads/geckodriver-v0.33.0-win64/geckodriver.exe");

        WebDriver driver = new FirefoxDriver();

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

        WebElement searchBox = driver.findElement(By.name("q"));

        searchBox.sendKeys( ...charSequences: "Selenium example");

        searchBox.sendKeys(Keys.RETURN);

        try {
            Thread.sleep( millis: 5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Page title: " + driver.getTitle());

        driver.quit();
    }
}
```

Figure 1. Sample Of Scripted Testing in Selenium

Scripted testing is ideal for:

1. *Regression testing*: Repeating pre-defined test cases to ensure existing functionality remains intact after application changes (Crispin & Gregory, 2009).
2. *Data-driven testing*: Executing tests with various data inputs to ensure functionality across different scenarios (Gundecha, 2015).
3. *API testing*: Automating API calls to validate responses and functionality (Beizer, 2003).

Pros:

1. *Repeatable and reliable:* Scripts guarantee consistent execution of test cases, providing reliable results (Patton, 2006).
2. *Time-saving:* Automates repetitive tasks, freeing up QA resources for other activities (Gregory & Crispin, 2009).
3. *Scalable:* Scripts can be easily scaled to test multiple applications and environments (Myers et al., 2011).
4. *Detailed logging:* Scripts provide detailed logs of actions and results, facilitating easy debugging (Bach, 2000).

Cons:

1. *Maintenance intensive:* Scripts require constant updates to reflect application changes, increasing maintenance overhead (Beizer, 2003).
2. *Brittle:* Scripts can break easily if the application's UI elements or behavior changes (Hetzel, 1988).
3. *Learning curve:* Requires knowledge of programming languages and Selenium framework (Nielsen, 1994).
4. *Limited flexibility:* Unfavorable for exploratory testing or adapting to unexpected behavior (Crispin & Gregory, 2009).

Non-Scripted Testing

Non-scripted testing, also known as exploratory testing or ad-hoc testing, involves manually testing the application using Selenium for recording and playback (Myers et al., 2011). However, unlike scripted testing, it does not rely on pre-defined scripts or test cases (Crispin & Gregory, 2019). Instead, testers use a more free-form approach, exploring the application's functionalities and using their judgment to identify potential issues (Hetzel, 1988).

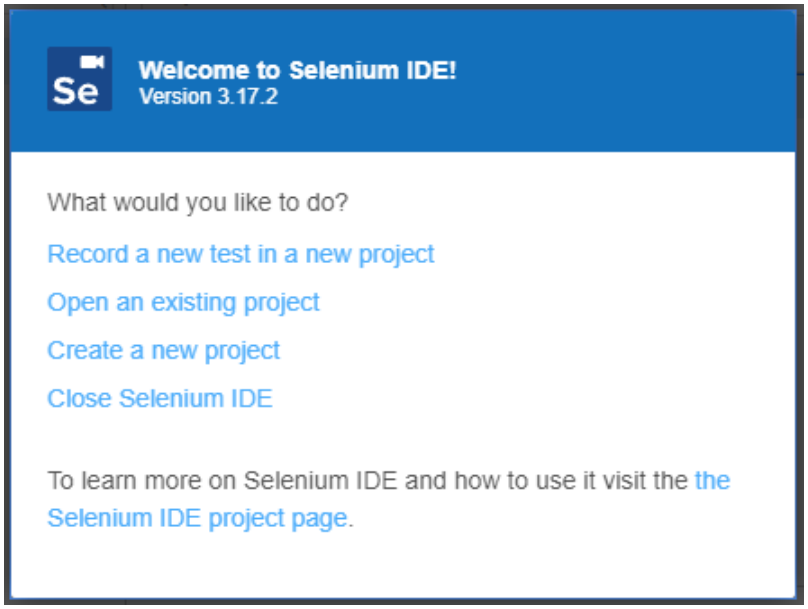


Figure 2. Selenium IDE for Non-scripted Testing

Non-scripted testing is particularly useful for:

1. *Exploratory testing:* Discovering new bugs and uncovering unexpected behavior (Hetzel, 1988). Testers can freely explore the application, trying out different features and combinations, which can help identify unforeseen issues or edge cases.
2. *Usability testing:* Evaluating user experience and identifying usability issues (Nielsen, 1994). By manually interacting with the application, testers can gain insights into how users interact with the interface and identify any usability problems.
3. *Ad-hoc testing:* Testing specific functionalities or scenarios that arise during manual testing (Beizer, 2003). This can be helpful for quickly testing out small changes or investigating specific problems that are encountered.

4. *Smoke testing:* Performing a quick test of the most critical functionalities before deploying a new build or release (Ji et al., 2019). Non-scripted testing can be a quicker way to ensure basic functionality before investing time in more formal testing.
5. *Learning about the application:* Gaining a deeper understanding of the application's functionality and behavior (Kaner et al., 1999). This can be especially helpful for new testers or when working with unfamiliar applications.

Pros:

1. *Flexible and adaptable:* Allows for exploring different functionalities and adapting to unexpected situations. Testers are not constrained by pre-defined test cases, allowing them to follow their intuition and investigate areas that seem interesting or suspicious (Alton, n.d.).
2. *Easy to learn:* No coding knowledge required, facilitating quick adoption. Even testers with limited technical expertise can participate in non-scripted testing, making it a valuable tool for agile environments (Crispin & Gregory, 2009).
3. *Effective for finding new bugs:* Enables testers to uncover hidden issues through exploratory testing. By exploring the application in a free-form manner, testers are more likely to find unexpected problems that may be missed by scripted testing (Hetzel, 1988).
4. *Reduces script maintenance:* No scripts to maintain, simplifying testing processes (Myers et al., 2011). This can save time and resources compared to scripted testing, which requires constant updates to reflect changes in the application.
5. *Promotes creativity and critical thinking:* Encourages testers to think creatively and use their problem-solving skills to identify potential issues (Crispin & Gregory, 2009). This can lead to a more comprehensive understanding of the application and its potential weaknesses.

Cons:

1. *Time-consuming:* Manual execution can be slower compared to automated scripts. This can be a drawback for testing large applications or complex functionalities (Daka & Fraser, 2014).

2. *Less repeatable:* Results can vary depending on the tester's approach and interpretation (Patton, 2006). This can make it difficult to compare results over time or between different testers.
3. *Difficult to document:* Capturing test steps and results can be challenging (Ji et al., 2019). This can make it difficult to share information and collaborate with other testers.
4. *Limited scalability:* Manual testing is not easily scalable for large test suites (Beizer, 2003). This can make it impractical for testing complex applications with a vast number of functionalities.

3.4 Static versus dynamic tests

Static Testing

Static testing analyzes the source code of your Selenium test scripts without actually executing them (Beizer, 2003). It helps identify potential issues like:

- Syntax errors: Missing commas, semicolons, or incorrect variable declarations.
- Logical errors: Unreachable code, infinite loops, and redundant test steps.
- Coding standards violations: Non-adherence to coding conventions.
- Potential performance bottlenecks: Repetitive code, unnecessary object creation, and inefficient data structures.

Benefits and weaknesses of static testing:

- Benefits:
    - Early defect detection: Finds issues early in the development process when they are easier and less expensive to fix (Myers et al., 2011).
    - Improved code quality: Ensures that your test scripts are well-structured, readable, and maintainable (Myers et al., 2011).
    - Reduced test execution time: Eliminates unnecessary test steps and optimizes code for better performance (Myers et al., 2011).
- Weaknesses:

- Limited scope: Can only detect issues within the test scripts themselves, not the application under test (Beizer, 2003).
- False positives: May flag issues that are not actually errors (Beizer, 2003).
- Requires expertise: Requires testers to be familiar with coding standards and best practices (Beizer, 2003).

Dynamic Testing

Dynamic testing involves executing your Selenium test scripts and observing the behavior of the application under test (Beizer, 2003). It helps detect a broader range of issues, including:

- Functional defects: Incorrect data displayed, buttons not working, unexpected navigation behavior.
- Performance issues: Slow loading times, memory leaks, and resource exhaustion.
- Accessibility issues: Elements not accessible to users with disabilities.
- Compatibility issues: Application not working properly on different browsers or operating systems.

Benefits and weaknesses of dynamic testing:

- Benefits:
  - Comprehensive testing: Covers a wider range of issues than static testing (Myers et al., 2011).
  - Real-world scenarios: Tests the application as it would be used by real users (Myers et al., 2011).
  - Integration testing: Allows testing of multiple components working together (Myers et al., 2011).
- Weaknesses:
  - Time-consuming: Requires executing test scripts which can take time (Beizer, 2003).
  - Maintenance overhead: Test scripts may need to be updated frequently as the application changes (Beizer, 2011).
  - Limited coverage: May not cover all possible scenarios and edge cases (Beizer, 2003).

Both static and dynamic testing play an important role in defect detection with Selenium (Myers et al., 2011). Deciding the right approach depends on your specific needs and project context (Beizer, 2003). By understanding the strengths

and weaknesses of each approach, you can develop a comprehensive testing strategy that ensures the quality and functionality of your web applications.

PART 4: Unit & Integration Testing

Unit Testing in software development is where different parts or components of a system are tested to check that they operate in the correct manner. Software verification focuses on the smallest software design unit- the module of the software component (Pressman, 2001; Shuaibu et al., 2019). Unit testing entails testing the smallest unit of the software (Sharma & Chandra, 2018). For instance, these include functions and methods. Furthermore, this kind of testing is referred to as components/module testing or white box testing (Shuaibu et al., 2019) and can be done at the same time for various components.

Once all modules have been unit-tested, they will undergo integration testing. Integration testing examines how two or more software modules or components interact with one another (Holling et al., 2016). It is an approach where codes are divided into separate segments but are tested as a group (Shuaibu et al., 2019). The individual components can affect each other in many ways, such as data loss across interfaces or when functions are combined, they may not produce the desired result. Thus, when all the components work individually, the challenge will be putting them all together (Pressman, 2001).

Unit test cases generated automatically help in test-driven development and give the developers a hand with suggested tests for detection of errors in their code (Tufano, 2020). These automation tools help reduce human intervention as well as the iterative process of testing different web applications (Sawant et al., 2021). This is one of the numerous tools such as Selenium. Selenium is automation software for testing web applications. Selenium WebDriver is a library that enables the automated control of web browsers. It also includes JUnit, a well-known testing framework for Java, among other tools. Unit and Integration testing can be applied using Selenium Web-driver with JUnit.

4.1 Functional analysis and use cases

Software project development starts with gathering the requirements upon which the scenario outlines how people can use the developed system. The scenarios that describe how the system will be used are called use cases (Pressman, 2001). It is a process of identifying, clarifying and organizing the system requirements using system analysis methodology (Brush, 2022). Interaction analysis are called functional analysis which is a type of use case that specifies processing functions and other operations to be carried out on the testable tool content (Pressman, 2001).

Use cases could be used to develop useful test cases for the Selenium testing tool that would test the expected functionality of web applications. Functional analysis shall help to understand what the system or the component should do in each use case.

4.2 Exception testing

An *exception* is a situation that arises while a program is being executed and interferes with the instructions' intended flow (Oracle, n.d.). The term exception is a shortened word for "exceptional event." Meanwhile, an exception handler is a code sequence created to catch and manage raised or thrown exceptions (Chang & Choi, 2016), and Exception testing is also known as Exception Handling Testing.

Exception handling or exception testing is the process of creating a system that can recognize and handle exceptional situations (Anjaneyulu, 2015). It helps improve a system, although it increases the costs of the testing phase. The exception-handling features in programming languages help programmers define, throw and catch exceptional conditions (Chang & Choi, 2016).

In the context of Selenium testing, exception testing is applied to ensure that test scripts gracefully handle unexpected issues that can or may occur during test executions. Exceptions in Selenium can be classified into two types: Checked Exceptions and Unchecked exceptions. Checked exceptions are exceptions handled while writing the code itself, while Unchecked exceptions are exceptions thrown at run time and are more catastrophic than Checked exceptions (Unadkat, 2023). Some of the common exceptions in Selenium WebDriver are NoSuchElementException, TimeoutException, SessionNotFoundException and many more.

Black box testing concentrates on the software's functional requirements, and this method of testing has several techniques, such as equivalence partitioning, boundary value analysis, comparison testing, and many more (Irawan et al., 2018). According to Santi et al. (2022), Black box testing with boundary value analysis and equivalence partitioning techniques is helpful in identifying malfunctioning features or functions so that the developer can fix or remove them. Another testing technique is the decision tables technique. Although an old technique, IT analysts and developers frequently use decision tables to manage, integrate, and execute complex logic more effectively (Sharma & Chandra, 2010). Decision tables are effective in finding faults in both implementation and specification. Later on, we will delve deeper into equivalence partitioning, boundary value analysis, and decision tables.

4.3 Equivalence partitioning

The equivalence partitioning technique assists in ensuring effectiveness by reducing the number of test cases. According to Pressman (2001), equivalence partitioning is a black-box test in which the input domain of the program is divided into data sets, from which the test cases are generated. It validates or invalidates the data grouped by the function by inspecting its input and output (Santi et al., 2022). A type or a class equivalency will be evaluated to define a test case with regard to an input condition, which can take the form of either numeric value, or a range, or set or even a Boolean condition (Santi et al., 2022). In the end, it divides the program’s input domain into equivalence classes based on the input values (Irawan et al., 2018). The resulting classes are utilized to draft test cases.

4.4 Boundary value analysis

Nidhra (2012) describes “Boundary Value Analysis”, as programming errors that happen at the boundaries of equivalency classes. Boundary Value Analysis is a testing method that is used alongside equivalence partitioning (Pressman, 2001). Pressman (2001) observed that error rates are higher near the input domain boundaries than in the center for unclear reasons. Because of this, Boundary Values Analysis technique was developed. This technique aims to uncover potential errors or issues at the edges or boundaries of the input values, as problems are often more likely to arise.

4.5 Decision tables

Decision tables are compact forms that allow test experts or design experts to express their knowledge in a readable manner (Nidhra, 2012; Beizer, 2003). Decision tables may be utilized when the program's logic or outcome depends on a set of decisions and guidelines that must be followed. The decision table consists of four areas: the condition stub, the condition entry, the action stub, and the action entry (Beizer, 2003; Nidhra, 2012; Sharma & Chandra, 2010). Accordingly, every column in a decision table represents a rule that outlines the circumstances in which the actions listed in the action stub will occur.

In summary, all of these parts contribute to the overall testing strategy for a software system. Ultimately, they ensure that software performs correctly under normal conditions through the functional analysis and use cases and that the exceptions are handled appropriately through the use of exception testing. The components ensure that the software is robust across different input conditions through equivalence partitioning and boundary value analysis, and the decision tables help correctly implement complex decision logic. Overall, these components contribute to the whole of unit and integration testing.

PART 5: System & User Acceptance Testing

5.1 Cross feature testing

Cross feature testing is a word referring to evaluating many features simultaneously (Hsueh & Ranasaria, 2008). Cross-browser testing or browser testing, is a quality assurance (QA) process that checks whether a web-based application, site or page functions as intended for end users across multiple browsers and devices (TSaar, 2016). The automation test cases can be done across various browsers, including Internet explorer, firefox, Chrome, and safari. However, it is possible to combine the TestNG framework to work concurrently for testing various browsers using the Selenium web driver. Such testing ensures uniformity on aspects such as image orientation, input positioning, JavaScript implementation, font size, and page alignment among operating systems. (TSaar, 2016)

5.2 Cause-effect graphing

A cause-and-effect graph is a black-box testing approach that visually depicts the link between a given outcome and all of the elements that impact it (Cause-Effect Graph, 2020). Selenium can be used to create a cause-effect graph though its different features. Recipes of selenium found in the internet can allow one to learn how to test common types of scenarios. For each different cause scenario, a corresponding recipe can be used to show how one can automate, and what its result is, allowing one to create a cause-effect graph.

5.3 Scenario testing

Scenario testing is a type of software testing in which actual situations are utilized instead of test cases to evaluate the software application. The goal of scenario testing is to evaluate end-to-end scenarios for a particular software problem. Scenarios make it easier to test and analyze complex problems from beginning to conclusion (Hamilton, 2021). In the selenium website there are recipes that can be used for pretty much any simple scenario, using different features of selenium.

5.4 Usability testing

Usability testing is a critical aspect of software development, ensuring that applications meet user expectations and preferences. While Selenium is widely known for its automation testing capabilities, it can also be used for usability testing, particularly for UI testing and cross-browser testing. Selenium automates web browser interaction, allowing test scripts to perform different user actions on the web application UI. This makes Selenium suitable for UI testing, as it can simulate user interactions and behaviors across different browsers and browser versions (Garima, 2023).

PART 6: Special Tests

6.1 Smoke tests

Homès (2012) defines a smoke test as: "A subset of all defined/planned test cases that cover the main functionality of a system component, to ascertain that the most crucial functions of a program work, but not bothering with finer details. A daily build and smoke test is among industry best practices."

The study by Ibrahim et al. (2019) explains that smoke testing, also called functional testing, is a software product focusing on the application's main features under development. It is further divided into three types, which are as follows:

Manual Smoke Testing

In this method, human software testers conduct smoke tests manually. This includes manually developing and updating test cases, along with test scripts are also written manually for new or existing features (Gillis, n.d.).

Automated Smoke Testing

Using software tools, automated smoke testing streamlines the testing process. These tools make the testing process more efficient by automatically providing and generating relevant tests (Gillis, n.d.).

Hybrid Smoke Testing

Hybrid testing is a combination of manual and automated smoke testing, where testers write test cases and then automate the tests using a tool (Gillis, n.d.).

Smoke Testing is a type of testing that is carried out after the new functions of the software are integrated with the old functions and ensures that the important functions of the software work properly (Banjarnahor & Istiyowati, 2022). Smoke testing is executed “before” any detailed functional or regression tests are run on the application being built.

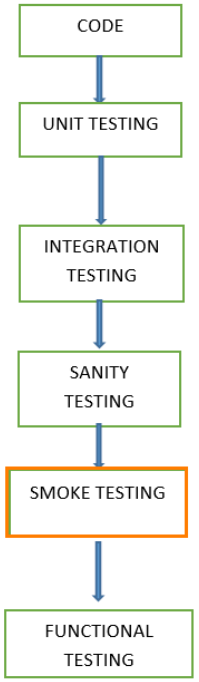


Figure 3. Smoke Testing Process

According to Banjarnahor & Istiyowati (2022), the primary objective is to detect and eliminate software issues early in development. This approach allows the Software Quality Assurance (SQA) team to save time installing and testing the software applications.

Hamilton (2022) provided a scenario where a new registration button is added in the login window, and the build is deployed with the new code. Testers will then perform smoke testing on this new build.

As identified by Sneha & Gowda (2017), the main goal of smoke testing is to test the dominant functions of the software but not to test all the features in depth. If the tests are passed, then testing will be continued to the next level. If not, then the testing is stopped, and will be informed to developers and ask them for a new build with the fix for the current failure.

Neglecting smoke testing in the early stages, as highlighted by Hamilton (2023), this may lead to more costly defects that will arise in later stages of development.

Once the build is deployed in Quality Assurance (QA) and smoke tests are successful, Hamilton (2023) emphasizes the time to proceed with functional testing. In cases where smoke tests fail, testers suspend testing until the issues in the build are solved. This approach ensures a systematic and efficient testing process.

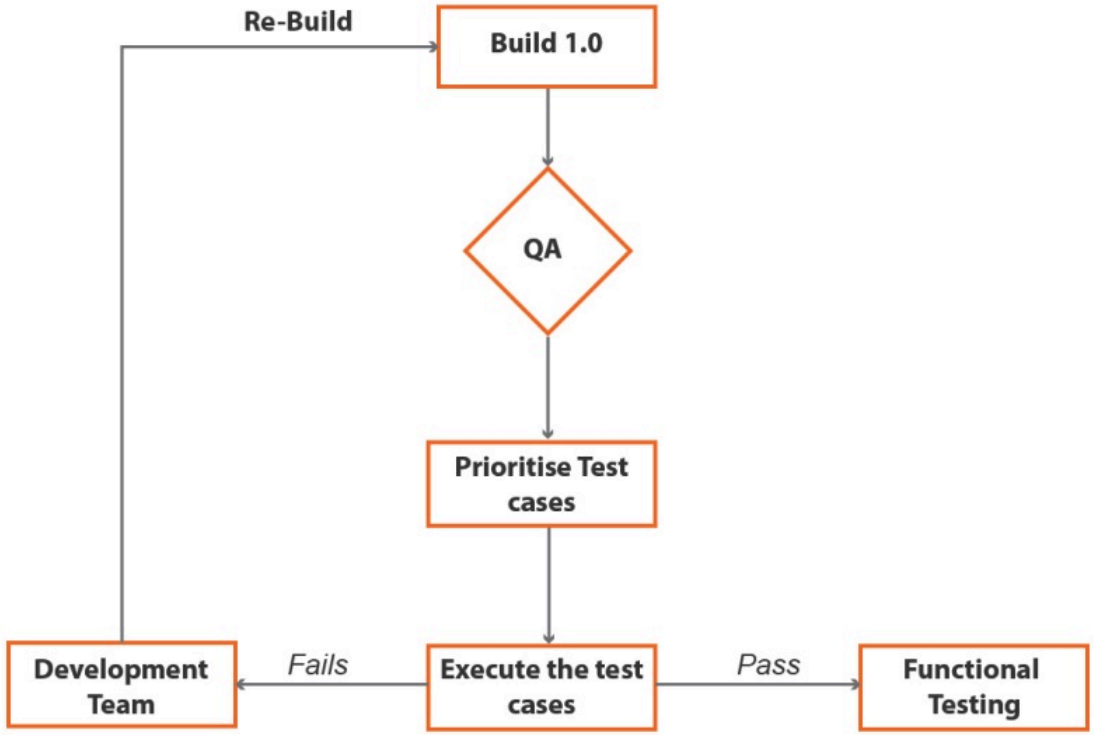


Figure 4. Smoke Testing Cycle

Sample Smoke Test Cases Documentation:

TEST ID.	TEST SCENARIO	DESCRIPTION	TEST STEP	EXPECTED RESULT	ACTUAL RESULT	STATUS
1	Adding item functionality	Able to add item to the cart	1. Select categories list. 2. Add the item to the cart.	The item should get added to the cart.	The item is successfully added to the cart.	Pass
2	Adding item functionality	Able to add item to the cart	1. Select categories list. 2. Add the item to cart.	Item should get added to the cart	Item is not getting added to the cart	Fail

Figure 5. Smoke Testing Test Cases Documentation

The open-source nature of Selenium allows it to automate smoke testing and run testing parameters on multiple web browsers (Gillis, n.d.). Selenium can automate control of browsers on various operating systems as well. However, as highlighted by Chandrasekharan et al. (n.d.), Selenium is exclusively designed for automating test cases in web applications on browsers

and is not usable for automating test cases for desktop or mobile applications. Selenium WebDriver, an advanced version of Selenium, addresses this limitation by directly interacting with browsers using their native support for automation, making it particularly suitable for browser-based automation tests (*Automated Smoke Testing: PhantomJS Vs. Selenium*, 2016).

Automated smoke testing using Selenium cuts down testing time drastically. According to Gillis (n.d.), manual testing can take half a day, depending on the number of smoke tests involved. Automated smoke testing with Selenium, especially using WebDriver, transforms this timeframe into a matter of minutes. Selenium WebDriver, conceived as a solution to the limitations of Selenium IDE, performs direct calls to web browsers, executing test scripts with browser-specific support and capabilities (Automated Smoke Testing: PhantomJS Vs. Selenium, 2016).

6.2 Performance tests

Performance testing is one of the non-functional testing types that tests software performance under all favorable and non-favorable conditions (Hooda & Chhillar, 2015). This testing occurs throughout all steps in the testing process and is specifically designed to assess the run-time performance of software within the context of an integrated system (Pressman, 2001).

Homès (2012) outlines two main facilities: load generation and test transaction measurement. Load generation is adept at simulating either multiple users or high volumes of input data. During execution, response time measurements are taken from selected transactions (Homès, 2012).

As for performance testing includes all timed parameters including load time, access time, run time, execution time, etc.; success rates, failure frequency, mean time between failures, and overall reliability software also included in this test (Hooda & Chhillar, 2015).

The methods of performance testing vary among these types and serve their distinct purposes:

Stress Testing

This type is performed to find and understand the upper limits of capacity within the system (Hooda & Chhillar, 2015).

Load Testing

This testing checks the application's ability to perform with the expected user loads. The objective is to know the performance bottlenecks before the software application goes live. (Hamilton, 2023)

Soak Testing

Also known as endurance testing, this testing determines if the system can sustain the continuous expected loads, identifying potential leaks detected by monitoring memory utilization (Hooda & Chhillar, 2015).

Spike Testing

This method is done by suddenly increasing the number of users or load generated by users by a substantial amount and observing the system's behavior (Hooda & Chhillar, 2015).

Scalability Testing

It determines the software application's effectiveness in "scaling up" to support an increase in user load. It helps plan capacity addition to software system (Hamilton, 2023).

Volume Testing

Large volumes of data are populated in a database, and the overall software performance is monitored (Hamilton, 2023).

The objective of the performance test is to demonstrate that the software system meets specific pre-defined performance criteria. Additionally, it can help compare the performance of two software systems. The primary intention is not to identify bugs but to eliminate bottlenecks that may arise due to coding errors and hardware issues, resulting in degrading the overall system performance (Hamilton, 2023).

In the study of Weyuker & Vokolos (2000), the usual approach to performance testing is to develop a benchmark workload that represents how the system will be used in the field. Running the benchmarks on the systems and comparing the results tacitly assumes that the system's behavior on the benchmarks will indicate how the system will behave when it is deployed because these benchmarks were specifically designed to be representative of real-world usage.

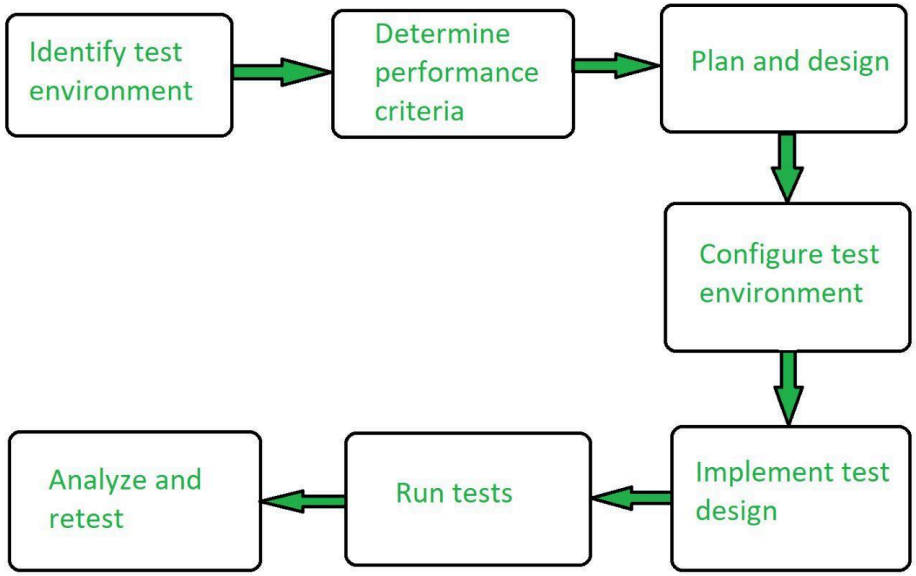


Figure 6. Performance Testing Process

Performance testing reveals what needs to be improved before the product goes to the market. Its purpose is to determine whether their software aligns with speed, scalability, and stability requirements when subjected to anticipated workloads (Hamilton, 2023).

Without performance testing, the software will likely suffer from several issues including slow performance during usage of multiple users, in inconsistencies across different operating systems, and poor usability (Hamilton, 2023). Software lacking performance testing is prone to gaining a negative reputation and falling short of expected sales goals.

Examples of Performance Test Cases	
Test Case No. 1	Verify the website's response time is not more than 4 seconds when 1000 users are accessing it simultaneously.
Test Case No. 2	Verify response time of the Application Under Load is within an acceptable range when the network connectivity is slow.
Test Case No. 3	Validate the maximum number of users the application can handle before it crashes.
Test Case No. 4	Validate the database execution time when 500 records are read and written simultaneously.
Test Case No. 5	Verify the CPU and memory usage of the application and the database server under peak load conditions.
Test Case No. 6	Verify the application's response time under low, normal, moderate, and heavy load conditions.

Figure 7. Example of Performance Test Cases According to Hamilton (2023).

According to the official documentation of Selenium, it presents that performance testing using Selenium and Selenium WebDriver is not recommended, not because it lacks the ability to carry out the task but because it is not suitable for the job and it is unlikely to get the expected results (*Performance Testing*, 2021).

While performance testing may seem ideal in the context of the user, a suite of WebDriver tests introduces a multitude of external and internal fragilities that are beyond one's control. For instance, browser startup speed, the speed of HTTP servers, the response time of third-party servers that host JavaScript and CSS, and the instrumentation penalty of the WebDriver implementation itself (*Performance Testing*, 2021). Based on Selenium's documentation, the variables introduced at these points will cause changes in results. Distinguishing the difference between the website's performance and external resources' performance becomes challenging. In addition, assessing the performance penalty sustained by using WebDriver in the browser, especially when injecting scripts, further complicates the task.

Another potential allure lies in the concept of "saving time," which carries out functional and performance tests at the same time. It is crucial to note that functional and performance tests pursue conflicting objectives. Functionality testing often requires testers to wait for loading times, demanding patience, which can cloud the performance testing results.

To enhance the website's performance, analyze the overall performance independently of environmental differences. This prospect entails identifying poor code practices and a breakdown of the performance of individual resources, such as CSS and Javascript, to know what to improve (*Performance Testing*, 2021).

6.3 Load/stress tests

Bayan & Cangussu (2006) emphasize that Load and Stress testing are important to guarantee the system's capability to support specified load conditions effectively from the excess use of resources. Load Testing involves assessing a system's performance under a given "load", representing the rate at which transactions are submitted to the system (Bayan & Cangussu, 2006).

The objective of load testing is to identify the maximum sustainable load that the system can handle. Notably, it reveals programming errors that remain hidden if the system executes with a small and limited workload. However these errors become apparent when the system operates under a heavy load—this is referred to as Load-sensitive faults (Bayan & Cangussu, 2006).

On the other hand, Stress Testing, as defined by Bayan & Cangussu (2006), subjects a system to an unreasonable load with the intention of breaking it. This test denies a system's essential resources (e.g., RAM, disk, interrupts, etc.) required to process a specific load, aiming to cause a system failure. In summary, stress testing is designed to evaluate a system's fault recovery capability (Bayan & Cangussu, 2006).

Load Testing and Stress Testing fall under the category of Performance Tests. Selenium's official documentation explicitly advises against using Selenium and WebDriver for performance testing, not because of their incapability, but because it is not optimized for the job and it may lead to unreliable results (*Performance Testing*, 2021). Loadview also agrees with this sentiment, emphasizing that Selenium and WebDriver alone are not recommended for executing load tests (*Selenium Load Testing Explained: Grid, JMeter & More*, n.d.).

Selenium Grid, another integral component of Selenium that is used to help reduce test runtime when executing multiple tests at one time, a practice known as parallel testing (*Selenium Load Testing Explained: Grid, JMeter & More*,

n.d.). Selenium Grid is composed of two elements called hubs and nodes. It operates by directing WebDriver requests and test commands (JSON) from hubs to nodes, where the test execution occurs.

However, in terms of load testing, limitations may arise when attempting to run load tests at scale. Large-scale load testing requires additional scaling, configuration, and maintenance requirements that open-source tools like Selenium Grid cannot support (*Selenium Load Testing Explained: Grid, JMeter & More*, n.d.).

6.4 Reliability testing

Reliability Testing is a software testing process that checks whether the software can perform a failure-free operation in a particular environment for a specified time period (Hamilton, 2023). Reliability testing aims to determine that the software product delivers its expected purpose (Homès, 2012). Hamilton (2023) outlines the key objectives for Reliability Testing, for instance- identifying patterns of recurring failures, finding the number of failures over time, discovering the root causes of failures, and conducting performance testing of various modules of software applications after fixing a defect.

As per Hamilton (2023), software reliability testing is divided into three types: Feature testing, load testing, and regression testing.

Feature Testing

Focuses on checking the features of each software operation's functionality, ensuring proper execution and minimal interaction between operations.

Load Testing

Load testing is conducted to check the performance of the software under the maximum workload.

Regression Testing

It is used to verify whether any new bugs have been introduced while fixing the previous bugs. It is conducted after every change or update of the software feature and their functionalities.

According to Unadkat (2023), A reliable test automation cycle relies considerably on the efficiency of Selenium scripts. Selenium simulates user interactions with clicks, input, and navigation because of this, developers can test the functionality of web applications and evaluate their responses to various loads and conditions. While not explicitly designed for testing reliability, it can be used with other tools to perform such testing (*Release Testing Explained: Best Practices and Examples*, n.d.).

Reliability testing is not inherently tailored for Selenium; however, Regression Testing is specifically crafted for effective utilization within the Selenium Framework. Deshpande (2023) noted that Selenium helps automate functional and regression test cases, which reduces the manual testing effort. Regression suites often include a huge number of test cases, and it takes time and effort to execute them manually whenever a code change has been introduced. Hence, organizations adopt the automation of regression test cases to streamline the testing process. Selecting the right automation framework depends upon the nature of the applications, technologies, testing requirements, and skill sets required for performing automation testing (Deshpande, 2023).

6.5 Acceptance tests

Acceptance Testing is a formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers, or other authorized entity to determine whether or not to accept the system (Homès, 2012).

This test will be performed when the system is handed over to the users/customers for that system from the developers' side. Acceptance testing aims to verify that the system is working rather than to find bugs/errors (Sneha & Gowda, 2017). When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. Acceptance testing can be conducted over a period of weeks or months (Pressman, 2001).

The Software Testing Fundamentals documentation states that Acceptance Testing can be categorized into multiple types (*Acceptance Testing*, 2022):

Internal Acceptance Testing

Also known as Alpha Testing, is also performed by members of the organization that developed the software but are not directly involved in the project.

External Acceptance Testing

It is performed by people who are not employees of the organization that developed the software.

Customer Acceptance Testing

It is performed by the customers of the organization that developed the software.

User Acceptance Testing (UAT)

Also known as Beta Testing is also performed by the end users (either existing or potential) of the software. They can be the customers themselves, the customers' customers, or the general public.

According to Selenium's official documentation, acceptance testing is the evaluation to determine whether a feature or system aligns with the customer's expectations and requirements. Selenium facilitates the execution of this testing directly on web applications by simulating everyday user actions (*Types of Testing*, 2023).

To sum up, these special tests have common themes that they follow. Each of these tests collectively contributes to ensuring the quality and reliability of a software application. Each of these testing methodologies serves each of their functionality: it provides a relation in the software development cycle, shares common themes of early detection, and each has the potential to utilize Selenium in their automation, especially in efficiency and time-saving aspects of Selenium remains consistent. While each testing type retains its distinct objectives, the main point is the assurance that the software aligns with user expectations, functions reliably, and performs optimally, contributing to delivering high-quality software products.

PART 7: Test Execution

Running a number of test cases to identify possible problems with the software in mind such as a web application is termed as test execution (Global App Testing, n.d). The tester has a tool called Selenium, particularly Selenium RC, which is essential in creating automated UI tests for Web app. Using Selenium RC, testers simulate what people do while they use the application, for instance, clicking a button or entering the text. The automation mentioned refers to any programming language such as Java, which ensures consistency and repetitiveness of user interface (UI) tests across multiple browsers including, Google chrome, Mozilla firefox, Safari, etcetera.

7.1 Test cycles

Commonly stated as a 'test cycle' for Selenium under the 'common test cycle definition'. Incorporating selenium into the testing cycle helps auto-generate and run repetitive tasks on the web application in order to verify its functionality across different browsers and platforms. Test cycles in Selenium typically involve:

- **Test Planning:** Specify the parameters of the website testing, identify the elements for testing, and devise test cases. It therefore serves as a guideline to the next set of actions in this case. Worth pointing out is that testing can also be expensive and therefore, testing everything is unrealistic regarding good testing practices and use of resources (Effective Test Planning and Implementation, n.d). Therefore, when planning tests, one should focus on what the project is all about, specific user scenarios and limitations.

- **Test Design:** testing design involves identification of the needed test cases. This involves writing Selenium scripts that would automate the interaction between selenium and the website under test. Testers have three choices of selection in Selenium that include Selenium WebDriver, Selenium Ide and Selenium Grid.

- **Test Execution:** This is the phase where all the test cases are executed on the software in question to ensure that it supports all the specifications needed. It entails run the scripts for Selenium against the web application. It should work on various browsers to guarantee compatibility with all browsers. As a result of this, there are generated reports and reports is the subsequent stage of the test cycle. One of these tools is Selenium IDE; which does cross-browser login testing for websites with ease.

- **Test Reports:** While conducting the testing, indicate the encountered problems in details and state how they were reproduced. In most cases, Selenium normally generates a report of test results containing any passages as well as fails. Nevertheless for a better understanding it's critical to consider the inclusion of reporting kits like

TestNG Reporter Log, JUnit Reporter Log, among others. With reporting tools, the organization will be able to spot bugs early on, follow up on web applications in terms of progress, and ease analyses.

7.2 Approaching a new feature

Feature refers to alteration of software that results in introducing new functionalities or altering old functionalities (What Is Feature Testing and Why is it important, 2023). Having new features and changes may lead to some issues with the current function, which could be really annoying (What is feature testing and why is it important, 2023). Hence as mentioned earlier, feature testing should be applied to guarantee proper operations of other features in existence when a new feature is introduced or a change occurs.

During implementation of a new feature or modification of an old feature, Selenium IDE facilitates the automation of browser actions and verification of their compatibility with actual product functionalities in Selenium. Functional test cases created without using scripting in Selenium IDE's playback tool. Thus, this makes it possible to test whether the new feature or changes are consistent with the features of the existing web application being tested.

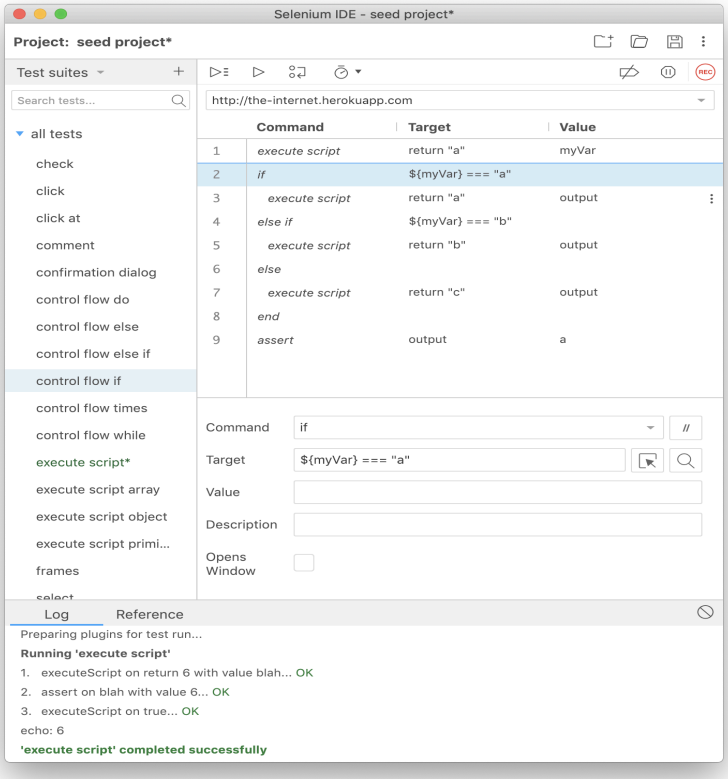


Figure 8. Selenium's IDE Record and Playback

Additionally, Selenium IDE can run on a variety of web browsers and platforms since the tester only needs to install it as a extension for the browser (Selenium, n.d). It also has an export as well as import mechanism, making the migration of test script accessible. As such, it qualifies as an effective tool for conducting feature testing in numerous web based applications.

7.3 Test data

Testers can also work with different test data such as Excel, XML, CSV, text files, etc., in Selenium (Vaidya, P:2023). In this case, excel format will be used by integrating Apache POI with Selenium as Apache POI enables a tester to import excel data using its Application Programming Interface to write codes in java that easily reads information from excel format. This facilitates automation of data entry during test operations such as login testing. Other software (OpenCSV) similar to Selenium can also be integrated for importing diverse types of data. Below is a sample code for the reading and printing of Excel information:

```java
String excelFilePath = "res/Sample_login_credentials.xlsx";

try (FileInputStream fileReader = new FileInputStream(excelFilePath)) {
    // Representation of Excel file
    XSSFWorkbook workBook = new XSSFWorkbook(fileReader);

    // Get the sheet by name
    XSSFSheet sheet = workBook.getSheet( name: "Sheet1");

    // Get the total number of rows and columns
    int totalRows = sheet.getLastRowNum();
    int totalColumns = sheet.getRow( rownum: 0).getLastCellNum();

    // For loop to print the data of the Excel
    for (int r = 0; r <= totalRows; r++) {
        XSSFRow thisRow = sheet.getRow(r);

        for (int c = 0; c < totalColumns; c++) {
            XSSFCell thisCell = thisRow.getCell(c);

            // Check if the cell is null before getting its value
            if (thisCell != null) {
                System.out.println(thisCell.getStringCellValue());
            } else {
                System.out.println("Cell is empty");
            }
        }
        System.out.println("================");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Figure 9. Example Code for Importing Excel Data using Apache POI

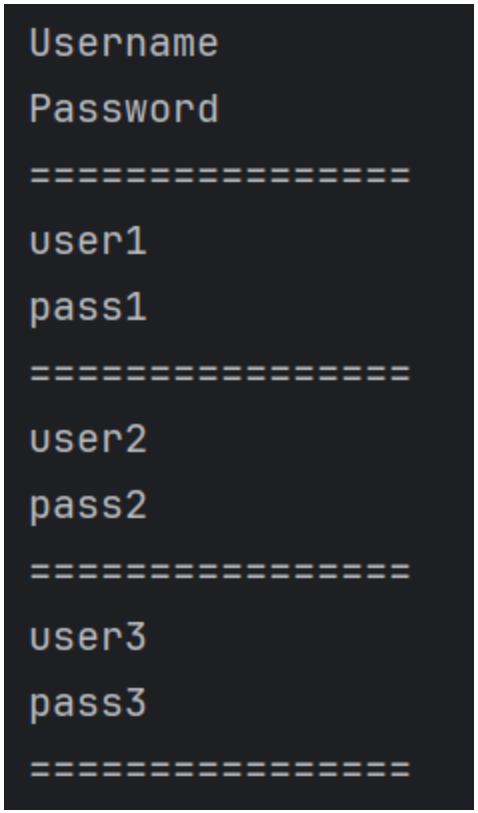


Figure 10. Print the Result of Imported Data

7.4 Tracking and reporting test results

R Indira (2001) says that a test summary should indicate whether a software product is ready to be released which in this case a web-based applications. It also serves as a point for enhancing web testing functionality. For instance, when one tries logging in to an E-commerce site, they fail due to one reason. In this case, the business will not only miss the specific sales from one user but also others who may have abandoned the cart. In Selenium, after testing the web application, it can give basic information on the successful and failed test cases like the examples below:

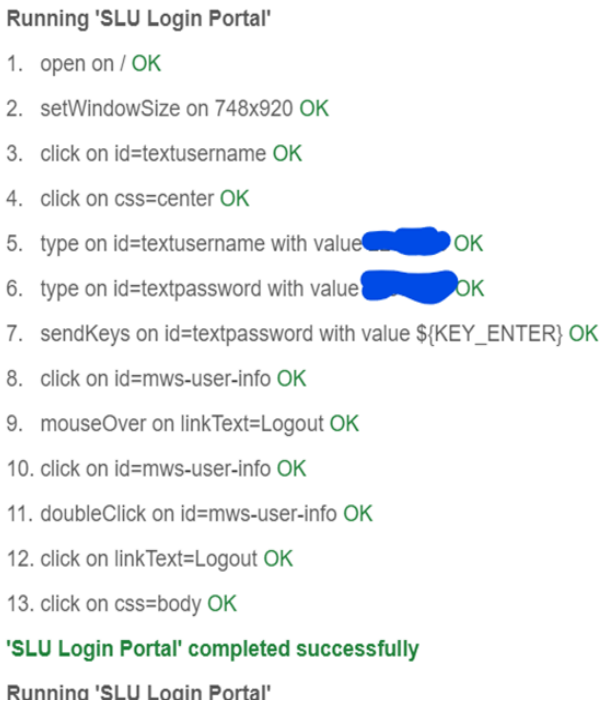


Figure 11. Successful Login Test Report using Selenium IDE

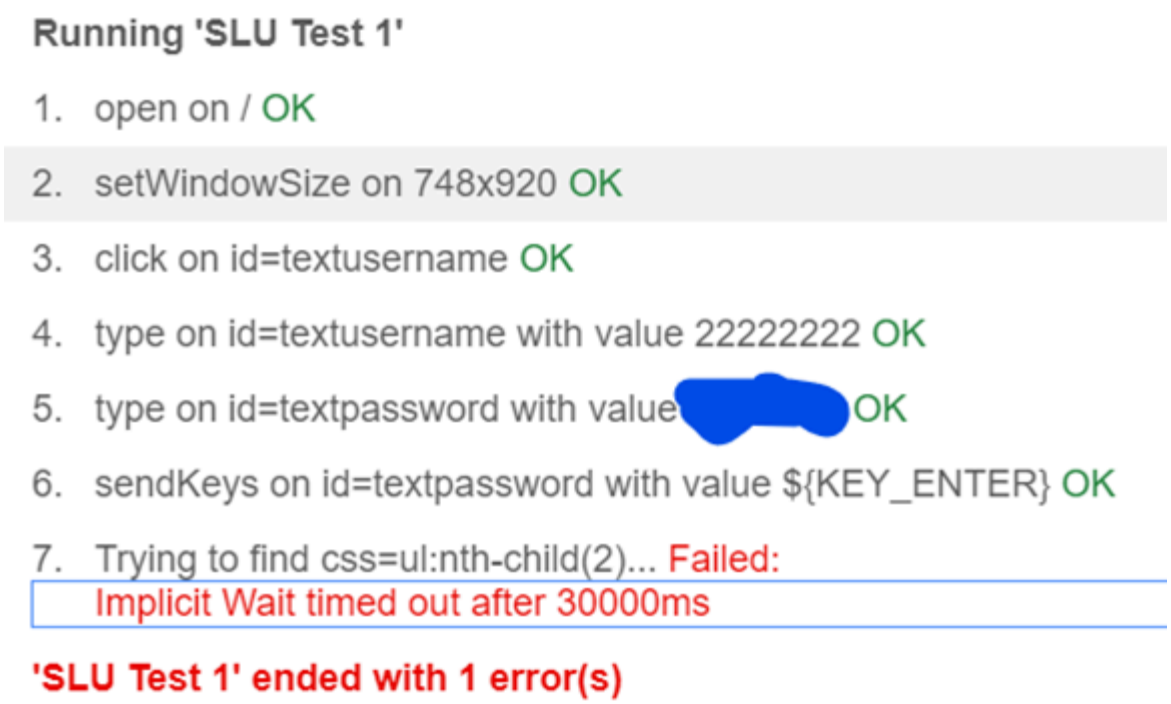


Figure 12. Failed Login Test Report using Selenium IDE

For instance, in the previous case, if a person is unable to log in to the portal and check on the status of his or her payments he or she would be unaware that I might still have balances for him or her. Moreover, if the log in does not work, they would instead go to the finance department themselves and check on their remittance status, something that is also inconvenient. In a bigger setting, it would be extremely hard for the students because they have to stand and wait for their turns. This explains the importance of testing the test summary since such bugs should not come up in future.

They have helped test testers into getting a clearer understanding of what happened during the test execution case. As an example, in this paper, we'll discuss using the TestNG reporting software, with IntelliJ IDEA. The steps in generating the test report are as follows:

Step 1: Install TestNG by adding it to the dependency inside the pom.xml of the created maven build system.

```
<!-- https://mvnrepository.com/artifact/org.testng/testng -->
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.8.0</version>
    <scope>test</scope>
</dependency>
```

Figure 13. Maven Dependency

Step 2: Install Selenium Java for Chrome driver dependency

```
<!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.15.0</version>
</dependency>
```

Figure 14. Selenium Java Dependency

Step 3: Create the test case/s

```
package org.example;

import org.testng.annotations.Test;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeClass;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterClass;

public class BrowserStackLoginTest {

    WebDriver driver;

    @BeforeClass
```

```
public void testSetup()
{
    System.setProperty("webdriver.chrome.driver", ".\\Driver\\chromedriver.exe");
    driver=new ChromeDriver();
    driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    driver.manage().window().maximize();

}

@BeforeMethod
public void openBrowser()
{
    driver.get("https://www.browserstack.com/");
    driver.findElement(By.id("signupModalButton")).click();
    System.out.println("We are currently on the following URL" +driver.getCurrentUrl());
}

@Test(description="This method validates the sign up functionality")
public void signUp()
{
    driver.findElement(By.id("user_full_name")).sendKeys("user_name");
    driver.findElement(By.id("user_email_login")).sendKeys("email_id");
    driver.findElement(By.id("user_password")).sendKeys("password");
    driver.findElement(By.xpath("//input[@name='terms_and_conditions']")).click();
    driver.findElement(By.id("user_submit")).click();

}

@AfterMethod
public void postSignUp()
{
    System.out.println(driver.getCurrentUrl());

}
```

```
    @AfterClass
    public void afterClass()
    {
        driver.quit();
    }

}
```

Figure 15. Login Test in Browserstack (Singh, 2023)

Step 4: Run the test case for the first and go to Edit Configurations and select an Output Directory to where the generated test report will be outputted. Then add a listener, which is the EmailableReporter that generates the test report.

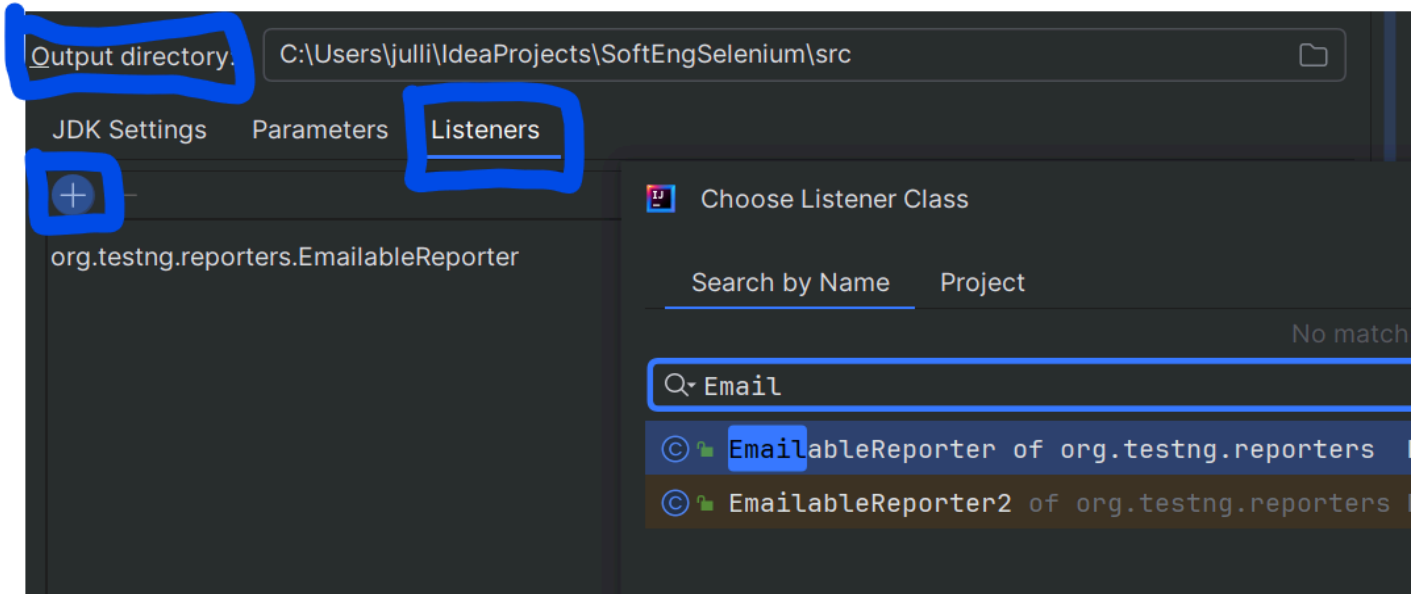


Figure 16. Output's Directory and Adding Listener

Step 5: The generated HTML file can be accessed from the Output Directory that has been set. Below is the sample output of the test summary.

Test	Methods Passed	Scenarios Passed	# skipped	# failed	Total Time	Included Groups	Excluded Groups
SoftEngSelenium	0	0	1	0	0.2 seconds		

Class	Method	# of Scenarios	Start	Time (ms)
SoftEngSelenium — failed (configuration methods)				
org.example.BrowserStackLoginTest	**testSetup**	1	1701775955173	155
SoftEngSelenium — skipped (configuration methods)				
org.example.BrowserStackLoginTest	**postSignUp**	1	1701775955359	0
	afterClass	1	1701775955361	0
	openBrowser	1	1701775955356	0
SoftEngSelenium — skipped				
org.example.BrowserStackLoginTest	**signUp** ("This method validates the sign up functionality")	1	1701775955357	0

Figure 17. Sample Test Report

PART 8: Defect Processing

No defect processing present in selenium. Selenium uses third-party applications for any Defect processing. The following would be a generalized idea of the topics that will be covered.

8.1 Defect life cycles

The identification and removal of software defects constitutes the basis of the software testing process, a fact which inevitably places increased emphasis on defect related software measurements (Lazić, 2010). A defect life cycle,

alternatively called bug life cycle, refers to a series of states that a bug or defect experiences in its entirety. All throughout the development process, defects are to be identified and resolved to ensure a quality software product. Once identified, developers are responsible for the documentation of these defects and provide fixes, so much so that developers would be able to replicate scenarios wherein they would reproduce said defect/bug. The number of states that a defect would go through varies depending on the tools and processes used during the testing phase of the software (Kumaresh & Baskaran, 2010). There are 5 workflow stages during a defect life cycle: (1) Defect Identification, (2) Defect Classification, (3) Defect Analysis, (4) Defect Prevention, and (5) Process Improvement.

a. Defect Identification

- Defects are found by preplanned activities specifically intended to uncover defects (Kumaresh & Baskaran, 2010). Defects are first identified in different stages of software development through code inspections, GUI review, unit testing, design review, etc. Once identified they are then classified using Orthogonal Defect Classification.

b. Defect Classification

- Orthogonal Defect Classification (ODC) is the most prevailing technique for identifying defects wherein defects are grouped into types rather than considered independently (Kumaresh & Baskaran, 2010). ODC are classified during two points: (1) When the defect was first detected – Opener Section, (2) When the defect got fixed – Closer Section.

  ODC methodology classifies each defect into orthogonal (mutually exclusive) attributes, some technical and some managerial (Kumaresh & Baskaran, 2010). These attributes provide the information to be able to read through volumes of data and arrive at patterns on which root-cause analysis can be done.

c. Defect Analysis

- Defect Analysis is using defects as data for continuous quality improvement (Kumaresh & Baskaran, 2010). This generally classifies defects into categories and identifies possible causes in order to create any possible improvements.

d. Defect Prevention

- Defect prevention is an important activity in any software project (Kumaresh & Baskaran, 2010). To simply put, this is to identify and prevent any defects from recurring.

e. Process Improvement

- The suggested improvements and preventive measures from the activities are to be implemented and be written in the existing manual.

8.2 Severity and priority of defects

Severity shows how bad the bug is and reflects its impact to the product and to the user, while priority can be decided on the basis of how frequently the defect occurs i.e. probability of occurrence of defect (Bhattad & Kothari, 2014). Under Severity there are 4 levels: (1) Trivial, (2) Minor, (3) Major, and (4) Critical.

a. Trivial

- Spelling, grammatical errors, etc. Defects of this level are mostly suggestions given to clients to make the application slightly better.

b. Minor

- Incorrect data, error messages, etc. Application can still continue, but with unexpected results.

c. Major

- Function is not working properly according to specifications. Application can still continue, but with severe defects.

d. Critical

- Inability to install/uninstall product, product often freezes or not run at all, Corrupted data is present, product abnormalities are present. Application is not suitable for release.

Under Priority defects there are 4 levels: (1) Low, (2) Medium, (3) High, and (4) Very High.

a. Low

- Would be fixed, but not top priority. Application can still be released before fix.

b. Medium

- Should fix, if there is ample time.

c. High

- Must be fixed before release of application

d. Very High

- Immediate fix, testing should not proceed until defect has been fixed.

8.3 Writing good defect reports

Bug reports are vital for any software development. They allow users to inform developers of the problems encountered while using a software (Zimmermann et al., 2010). They contain detailed information about failure of a code, or a faulty behavior present in the software. Along with the observed and expected behavior, bug reports often contain the steps to reproduce the bug (Chaparro et al., 2019). Unfortunately, in many cases, the bugs are unclear, incomplete, and/or ambiguous. So much so that developers are often unable to replicate the problems, let alone fix the bugs in the software (Chaparro et al., 2019). Creating a good defect report is crucial in creating effective communication between tester and developer. By providing quality defect reports, developers are able to better understand the bugs that need to be fixed. There are 17 sections that a Defect Report has: (1) Defect Title/Summary, (2) Defect ID, (3) Project Name, (4) Version Number, (5) Environment, (6) Steps to Reproduce, (7) Expected Results, (8) Actual Results, (9) Visual Aid, (10) Test Data, (11) Severity, (12) Priority, (13) Assignee, (14) Reporter, (15) Date Reported, (16) Status, and (17) Comments

a. Defect Title/Summary

- It offers an introductory phrase depicting the defect. It should be clear with no additional elaboration on why the products have become defective.

b. Defect ID

- It provides a distinct identifier for the defect and helps in tracking it during the course of development and testing.

c. Project Name

- This will indicate the project in which the defect was discovered.

d. Version Name

- This would show the project which specific version of build the fault was found. This would enable the developer to return on a particular update level of the project.

e. Environment

- These include the specifications used by the tester. Browser, browser version, operating system, hardware and so on. It is very essential to attempt to reproduce defects.

f. Steps to Reproduce

- Describes the procedure followed in discovering the defect.

g. Expected Results

- The above entails what is supposed to be done by the module. This serves to establish the foundation for understanding of the right functional operation.

h. Actual Results

- This describes the behavior during testing. States the results of testing to the developers and compares the expected results from the actual testing results.

i. Visual Aid

- visual aid can provide developers additional context and understanding on the defect much more easily.

j. Test Data

- Specifies any relevant test data used during testing that might have contributed to the defect.

k. Severity

- Assigns the Severity level of the defect and how impactful it is to the application.

l. Priority

- Assigns the Priority level of the defect on how urgent it is to be fixed. This would also determine the order of defects that should be addressed.

m. Assignee

- Specify which developer or team of developers are to be responsible for fixing the defect. This ensure that the team handling the defect would be the same team handling that specific functionality

n. Reported By

- This would provide the name of the tester who discovered and reported the defect. This would also provide easier communication among testers and developers to whom they need to contact.

o. Date Reported

- This includes the date that the defect was reported. Important for tracking the history of the defect for future use.

p. Status

- Indicate the current status of the defect, whether the defect has been resolved, in progress of being fixed, or open for fixes. This tracks the resolution of the defect.

q. Comments

- This would provide additional information that would be useful for the developers. This might include the behavior of other functions to the defects, some additional details of the defect, etc.

In each section, there can be varying levels of detail depending on the project being developed, specificity of information, testing teams and developer teams, etc.

8.4 Efficient use of defect tracking tools

A bug tracking system is a software application that is designed to help quality assurance and programmers keep track of reported software bugs in their work (Singh, 2013). Provides convenience to both testers and developers to track all reported bugs and defects present and resolved. Stakeholders can take advantage of these tools by generating productivity reports of programmers and attitude towards risk handling. Although it is not a perfect criteria to measure efficiency of a programmer yet can provide a coverall idea of productivity ("Study of Various Bug Tracking Tools," 2020). There are several tips on how to effectively and efficiently use defect tracking tools: Standardize the Process, Prioritize and Categorize, Integration with Development Tools, Communication and regular updates, Use Reports and Metrics, and Provide Training to the Team.

a. Standardize the Process

- Define and create standard workflows from the point where the defect is discovered, up to the time the defect is resolved. By following the defined workflow, consistency will be maintained throughout the process.

b. Prioritize and Categorize

- Prioritizing issues and defects based on how hard the software problem would help developers focus on critical errors and defects first. Additionally, categorizing would help classify each defect for better organization and tracking of defects.

c. Integration with Development Tools

- By combining the development tools with defect tracking tools can provide an efficient transition when working between testing and developing.

d. Communication and regular updates

- Regularly updating the status of defects and communicating it between tester and developer would provide detailed information throughout the whole development process of the project.

e. Use Reports and Metrics

- Reporting on the defects found as well as assessing the software's performance via metrics can help identify and review the areas for improvement during the development process and the testing process.

f. Provide training to the team

- Ensure the team is trained on how to use the defect tracking tool effectively. By providing support and documentation, this would help assist the team on effectively using the features of their chosen defect tracking tool.

PART 9: Test Automation

9.1 Test automation approaches

Test Automation, defined as the utilization of software tools to automate the execution of tests and comparison of actual outcomes with expected results (Myers, 2011), offers significant advantages. This approach can significantly reduce time and resource expenditure, while also enhancing the accuracy and comprehensiveness of testing. In the context of Selenium, automated testing empowers testers to automate browser interactions, replicating user behavior and verifying application functionality. This capability proves to be a valuable asset in guaranteeing the quality and dependability of web applications. Within Selenium, three primary approaches to test automation exist.

1. Keyword-Driven Testing (KDT)
    a. This approach involves writing tests using human-readable formats such as plain text or spreadsheets. A framework then translates these keywords into Selenium commands for execution. This approach offers ease of use and learning for beginners, but may lack flexibility and maintainability compared to others (Easy, 2021).
2. Data-Driven Testing (DDT)

a. This approach separates test data from test logic. Test logic is written in a programming language like Python or Java, while test data is stored separately in a format like CSV. DDT promotes flexibility and maintainability compared to KDT, but requires programming knowledge (Hamilton, 2023).

3. Page Object Model (POM)
    a. This approach involves creating a dedicated class for each page in the web application. Each class encapsulates all elements and functionalities of the corresponding page. POM can enhance test modularity and maintainability, but requires more upfront setup time (GeeksforGeeks, 2019).

9.2 Automation process

Selenium, created in 2004 by Jason Huggins, automated web testing with the JavaScriptTestRunner. While effective for web applications, its limitation excludes desktop and mobile apps. Alternatives like Appium and HP's QTP address this gap in Selenium's coverage.

To start the automation process, the first thing to do is to install Selenium in the browser.

Step 1 - Open the Browser

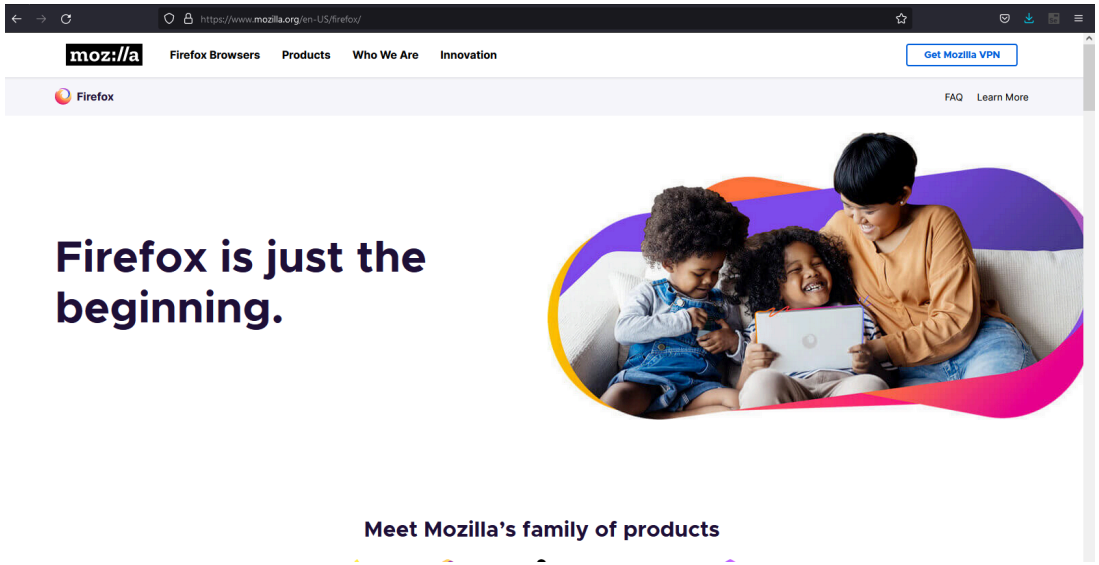


Figure 18. Opening the Browser

Step 2 - Click on the menu in the top right corner and click on add-ons.

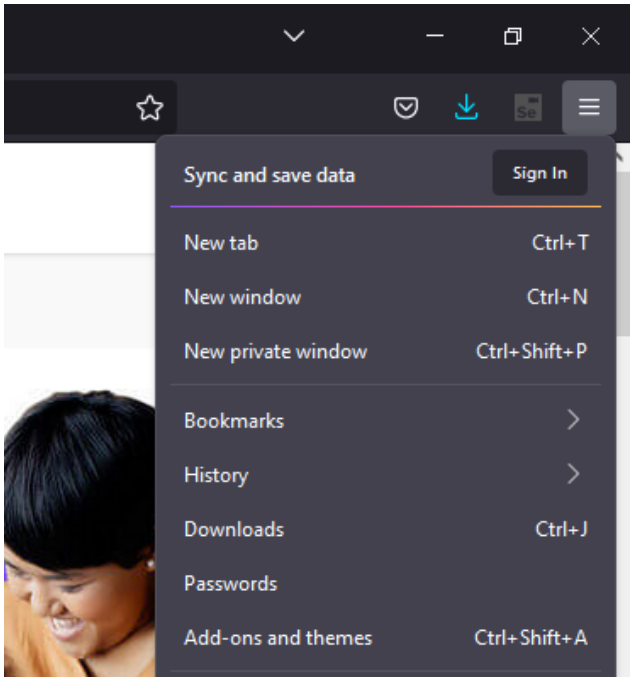


Figure 19: Menu Settings

Step 3 - Click on find more and type “Selenium IDE”

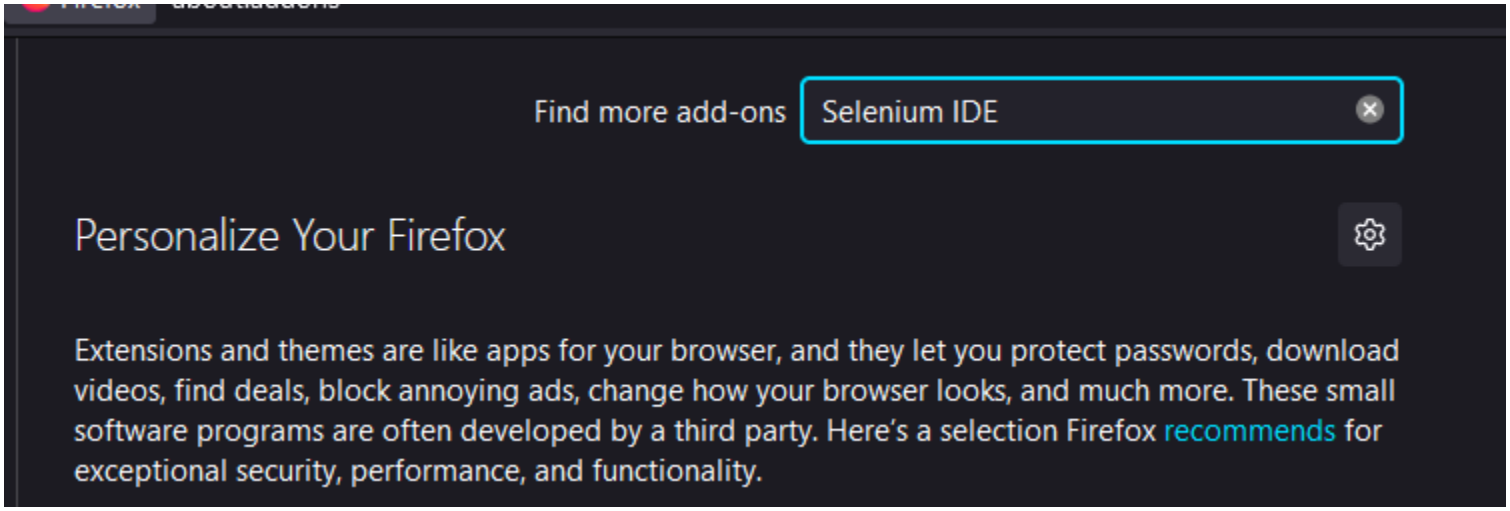


Figure 20

Step 4 - Click on Add to “browser”

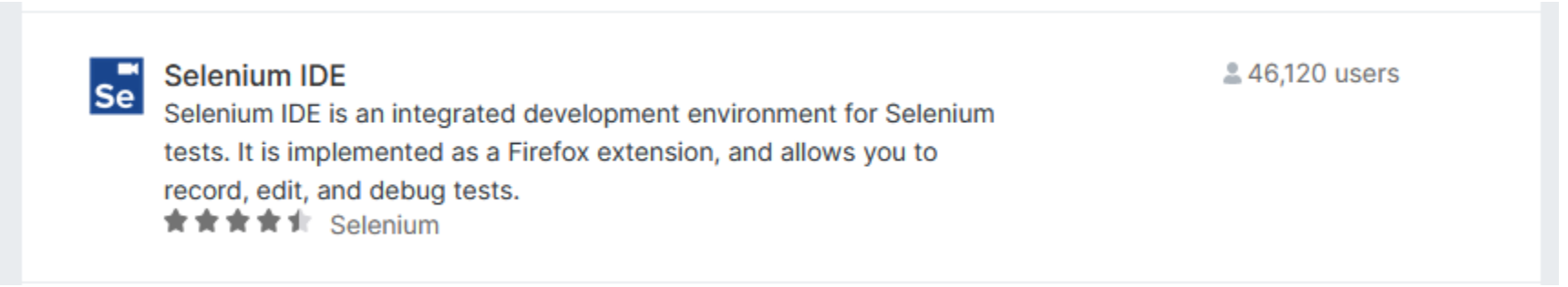


Figure 21

After installing Selenium on in the browser, it is now possible to record a test of any website.

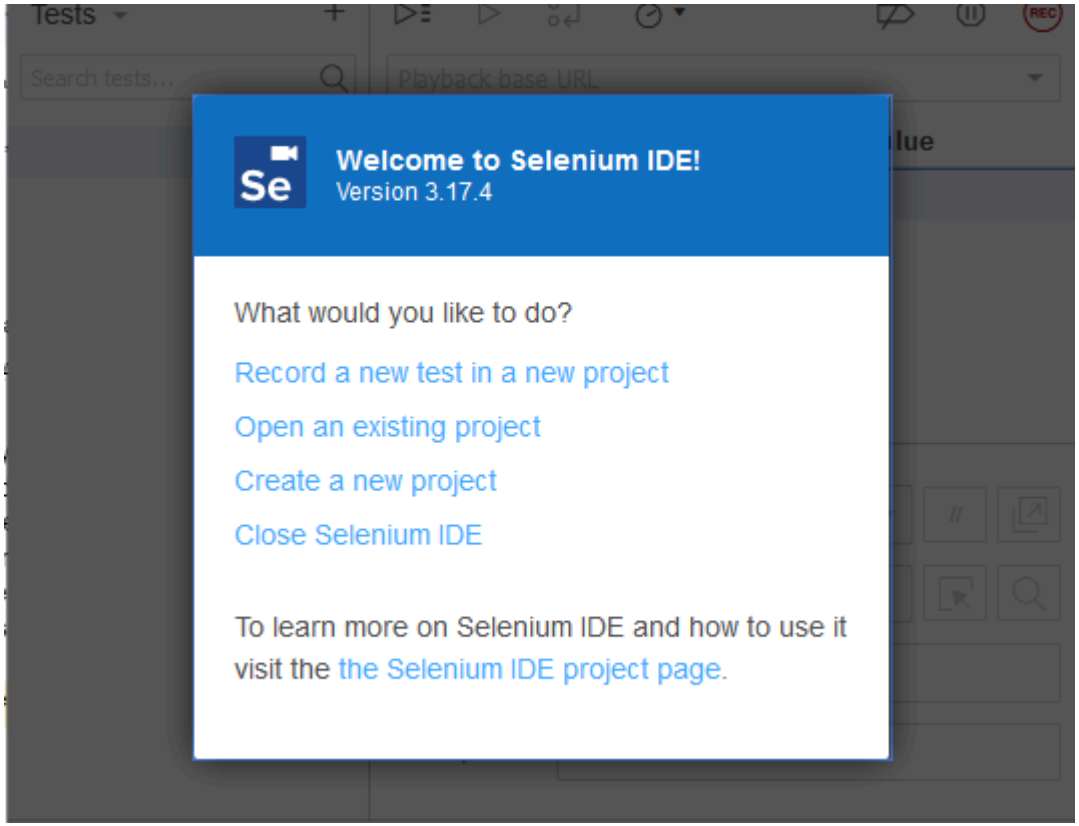


Figure 22.

To record a test, start by entering a website.

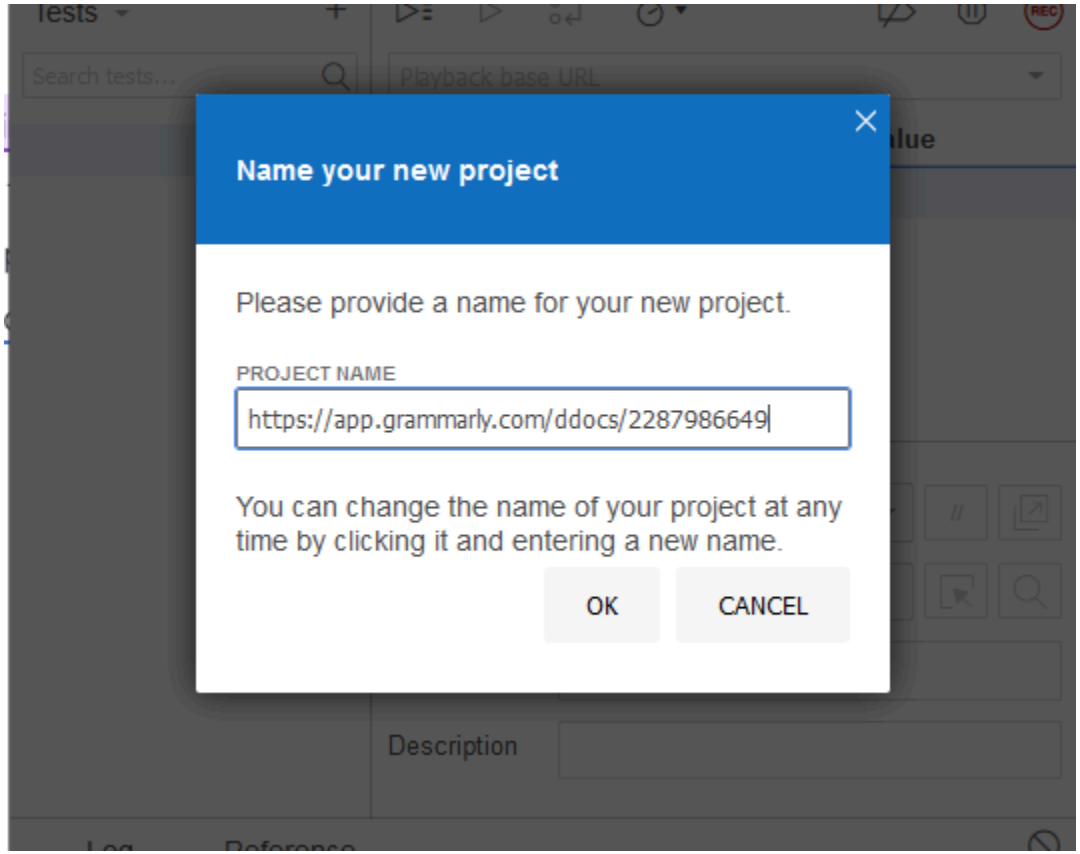


Figure 23.

In the other window of the Selenium IDE, is where actions like saving one's work, playback, run or pause the test.

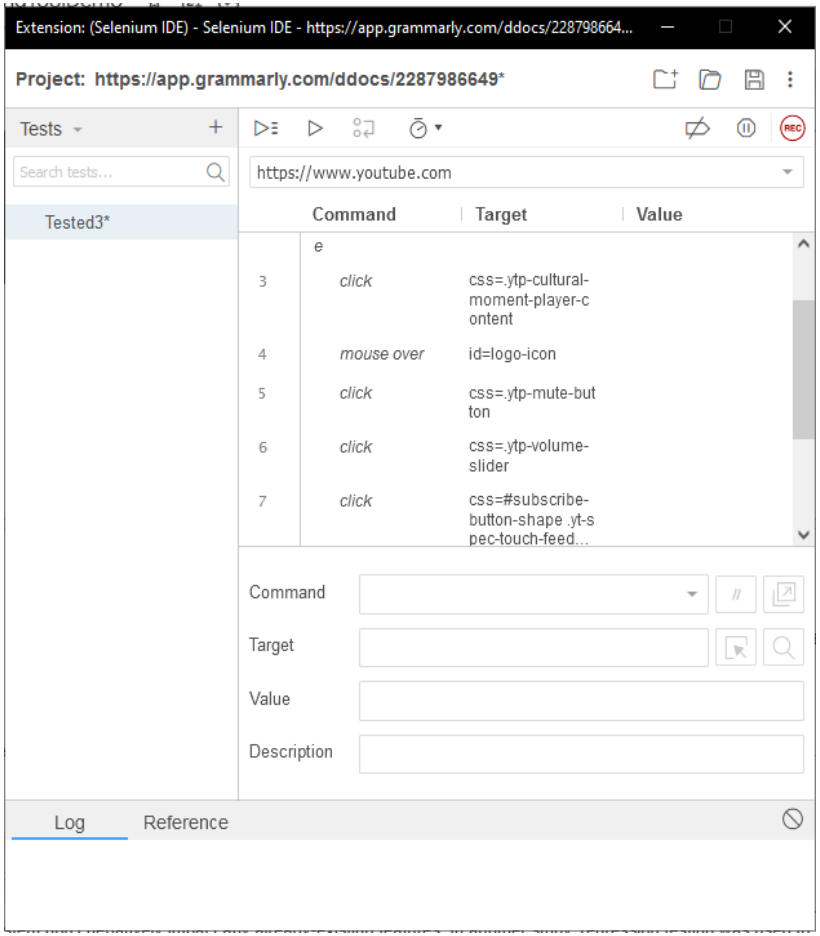


Figure 24. Selenium IDE Playback

9.3 Unit test frameworks

Unit Testing Frameworks are simply software tools to support writing and running unit tests, including a foundation on which to build tests and the functionality to execute the rests and report their results (Hamill, 2004). Unit testing involves testing the most basic units of code, the Unit Testing tools are used to aid unit testing process. Practically, unit testing tools are the most used tools to automate tests and are easily integrable as a framework within a development environment such as NetBeans (Umar & Zhanfang, 2019). Unit testing tools are used to tests the correct working of a particular unit (or method) as well as to check code structure and ensure correct observance of programming practice and also ensure correct working of individual units (Umar & Zhanfang, 2019). Selenium supports various unit testing frameworks based on multiple programming languages like Java, C#, PHP, Ruby, Perl, JavaScript, and Python such as JUnit, JEST, PyTest, NUnit, MSTest, PerlUnit, Test:Unit, and many more. These frameworks are used for executing test scripts on web applications across different platforms such as Windows, MacOS, and Linux.

9.4 Automation of regression testing

According to Ali and Arcolas (2019), Regression testing is a kind of testing used to confirm that newly pushed changes to the system don't negatively impact any already-existing features. In another study, regression testing was used to ensure that new defects were not introduced into the software due to changes made to its execution environment (Ali et al., 2019). All phases and levels of development involve a great deal of repetitive testing, and for large and complex systems, accuracy in regression test scoping is essential. Manual execution of testing deals with a great amount of time spent, and the cost of running these tests is high. Due to these, automated testing was developed. Automated regression testing greatly reduces the time spent by testers to perform these repetitive and mundane tests and allows them to work on more critical tests (Ali & Arcolas, 2019).

9.5 Minimizing test automation maintenance

According to Jash Unadkat, Technical Content Writer at BrowserStack the best way to minimize test automation maintenance is through following these best practices.

1. **Locator Strategy Mastery:** Locators should be perfect in their operation because they ensure correct identification of elements failure being avoided for this reason. Consider locators guides by Selenium for matching strategies suitable of different testing cases.

2. **Page Object Model (POM):** Use product oriented modules (POM) to group webpages into class file and minimize repeated codes for clarity, saving memory, data access and improving performance. This makes this design patterer lessen the test maintenance as well as encourage code reusability during a UI change.

3. **Real Device Testing:** For accurate test results, run Selenium tests on real devices through platforms such as BrowserStack. Cloud Selenium Grid delivers 3000+ real browsers for exhaustive cross-browser and device tests with CI/CD integration.

4. **Screenshots on Failure:** Immediately take screenshots upon test failures towards fast problem solving . This makes it easy for testers to identify such bugs and with great ease using BrowserStack's Cloud Selenium Grid.

5. **Browser Compatibility Matrix:** The usability profile, usage metrics, user preference data, and product performance reports can be utilized to develop an easy-to-use browser compatibility matrix. Cutting down on test coverage to suitable browser-OS combinations improves efficiency.

6. **Wait Commands Optimization:** Using Implicit/Explicit wait commands instead of Thread.sleep() to handle page load times efficiently. Through this approach, the test execution becomes fast and reliable as it adjusts to varying loading speeds.

7. **Strategic Test Case Planning:** Prepare an extensive test plan prior to automation incorporating logical scenarios from the end-user's point of view. Strategic planning ensures that there are no bottlenecks and that testing is done all the way up in the develop cycle.

8. **Test Case Identification and Prioritization:** Run first high priority tests like login functionality. Articulating and concentrating on most important test cases improve test coverage as critical issues are handled first in the testing flow.

PART 10: Appendices

References

Acceptance Testing. (2022, August 29). Software Testing Fundamentals. https://softwaretestingfundamentals.com/acceptance-testing/#google_vignette

Accessibility Testing With Selenium Webdriver with Code Example. (2023, June 25). Software Testing Help. https://www.softwaretestinghelp.com/accessibility-testing-with-selenium/

Ahmad, N. (2023, February 20). Retesting Tutorial: a comprehensive guide with examples and best practices. dzone.com. https://dzone.com/articles/retesting-tutorial-a-comprehensive-guide-with-exam

Ali, N. B., Engström, E., Taromirad, M., Mousavi, M. R., Minhas, N. M., Helgesson, D., Kunze, S., & Varshosaz, M. (2019). On the search for industry-relevant regression testing research. Empirical Software Engineering, 24(4), 2020–2055. https://doi.org/10.1007/s10664-018-9670-1

Ali, S., & Arcolas, N. K. P. (2019). Automated Regression Tests and Automated Test Optimisation for GETRV. 5th International Conference on. . . ResearchGate. https://www.researchgate.net/publication/343178597_Automated_Regression_Tests_and_Automated_Test_Optimisation_for_GETRV_5th_International_Conference_on_Advances_in_Computer_Science_and_Information_Technology

Alton, M. (n.d.). *Rapid software testing Methodology*. Satisfice, Inc. https://www.satisfice.com/rapid-testing-methodology

Anjaneyulu, G. (2015). Automated exception handling in software testing. ResearchGate. https://www.researchgate.net/publication/286856587_Automated_exception_handling_in_software_testing

Ashawami. (2022, November 10). User Journey Test - Ashawami - Medium. *Medium*. https://medium.com/@ashawami4/user-journey-test-394e10f83cae

Automated Smoke Testing: PhantomJS vs. Selenium. (2016, November 23). Cabot Technology Solutions. https://www.cabotsolutions.com/automated-smoke-testing-phantomjs-selenium

Banjarnahor, M., & Istiyowati, L. (2022). *Smoke Automation and Regression Testing on a peer-to-peerlending Website with the Data-DrivenTesting Method*. http://jurnal.iaii.or.id/index.php/RESTI/article/view/4220/638

Baskirt, O. (2022, February 6). *Cross browser testing in Selenium examples*. Software Test Academy. https://www.swtestacademy.com/cross-browser-testing-in-selenium/

Bayan, M. S., & Cangussu, J. W. (2006). *Automatic Stress and Load Testing for Embedded Systems*. https://ieeexplore.ieee.org/abstract/document/4020172

Beizer, B. (2003). Software testing techniques. Dreamtech Press.

Bharati, N. (2023, February 14). Top 5 Selenium reporting Tools | BrowserStack. BrowserStack. https://www.browserstack.com/guide/top-selenium-reporting-tools

Bhardwaj, D. (2023, November 17). Test Execution tutorial: A comprehensive guide with examples and best practices. https://www.lambdatest.com/learning-hub/test-execution

Bhattad, B., & Kothari, A. (2014). Study of defects, TestCases and testing challenges in website projects using manual and automated techniques. Computer Science & Information Technology (CS & IT). https://doi.org/10.5121/csit.2014.4907

Chandrasekharan, L., Wickramasinghe, S., & Rajora, H. (n.d.). *What is Smoke Testing? - A Detailed Guide*. Testsigma. https://testsigma.com/smoke-testing/

Chang, B., & Choi, K. (2016). A review on exception analysis. Information & Software Technology, 77, 1–16. https://doi.org/10.1016/j.infsof.2016.05.003

Chaparro, O., Bernal-Cárdenas, C., Lü, J., Moran, K., Marcus, A., Di Penta, M., Poshyvanyk, D., & Ng, V. (2019). Assessing the quality of the steps to reproduce in bug reports. ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. https://doi.org/10.1145/3338906.3338947

Crispin, L., & Gregory, J. (2009). *Agile testing: A Practical Guide for Testers and Agile Teams*. Pearson Education.

Carty, D. (2018, November 20). cross-browser testing. SearchSoftwareQuality. https://searchsoftwarequality.techtarget.com/definition/cross-browser-testing

Cause-Effect Graph Technique in Black Box Testing - javatpoint. (n.d.). Www.Javatpoint.Com. https://www.javatpoint.com/cause-and-effect-graph-technique-in-black-box-testing

Daka, E., & Fraser, G. (2014). A Survey on Unit Testing Practices and Problems. *IEEE Access*. https://doi.org/10.1109/issre.2014.11

Deshpande, P. (2023, February 13). *Regression Testing with Selenium: Tutorial*. BrowserStack. https://www.browserstack.com/guide/regression-testing-with-selenium

Effective test planning and implementation. (n.d.). smartbear.com. https://smartbear.com/test-management/test-plan/effective-test-planning/

Easy, S. (2021, July 12). *Keyword driven framework in Selenium WebDriver*. Scientech Easy. https://www.scientecheasy.com/2020/01/keyword-driven-framework-in-selenium.html/

Hetzel, W. C. (1988). *The complete guide to software testing*.

Irawan, Y., Muzid, S., Susanti, N., & Setiawan, R. R. (2018). System Testing using Black Box Testing Equivalence Partitioning (Case Study at Garbage Bank Management Information System on Karya Sentosa). Proceedings of the the 1st International Conference on Computer Science and Engineering Technology Universitas Muria Kudus. https://doi.org/10.4108/eai.24-10-2018.2280526

Jash, U. (Feb 2023) Best Practices for Selenium Test Automation https://www.browserstack.com/guide/best-practices-in-selenium-automation

Jenkins User Documentation. (n.d.). Jenkins User Documentation. https://www.jenkins.io/doc/

Ji, S., Li, B., & Zhang, P. (2019). Test case selection for All-Uses Criterion-Based Regression Testing of composite service. *IEEE Access*, *7*, 174438–174464. https://doi.org/10.1109/access.2019.2957220

Kumaresh, S., & Baskaran, R. (2010). Defect analysis and prevention for software process quality improvement. International Journal of Computer Applications, 8(7), 42–47. https://doi.org/10.5120/1218-1759

García, B., Kloos, C. D., Alario-Hoyos, C., & Muñoz-Organero, M. (2022). Selenium-Jupiter: A JUnit 5 extension for Selenium WebDriver. Journal of Systems and Software, 189, 111298. https://doi.org/10.1016/j.jss.2022.111298

GeeksforGeeks. (2019, July 16). *Page Object Model POM*. https://www.geeksforgeeks.org/page-object-model-pom/

Get started with GitLab CI/CD | GitLab. (n.d.). https://docs.gitlab.com/ee/ci/

Gillis, A. S. (n.d.). *What is Smoke Testing? | Definition from TechTarget*. TechTarget. https://www.techtarget.com/searchsoftwarequality/definition/smoke-testing

Global App Testing. (n.d.). Test execution: how to get yours right. https://www.globalapptesting.com/test-execution#:~:text=Test%20execution%20is%20the%20process,issues%20your%20software%20could%20have.

Gundecha, U. (2015). *Selenium Testing Tools Cookbook Second Edition*. Packt Publishing.

Hamill, P. (2004). Unit test frameworks. https://openlibrary.org/books/OL3435021M/Unit_test_frameworks

Hamilton, T. (2021, October 8). What is Test Scenario? Template with Examples. Guru99. https://www.guru99.com/test-scenario.html

Hamilton, T. (2023, October 14). *What is Reliability Testing? (Example)*. Guru99. https://www.guru99.com/reliability-testing.html

Hamilton, T. (2023, October 14). *Load Testing Tutorial: What is? How to? (Examples)*. Guru99. https://www.guru99.com/load-testing-tutorial.html

Hamilton, T. (2023, October 28). *Performance Testing Tutorial – Types (Example)*. Guru99. https://www.guru99.com/performance-testing.html

Hamilton, T. (2023, November 18). *What is Smoke Testing?* Guru99. https://www.guru99.com/smoke-testing.html

Hamilton, T. (2023, November 25). *Software testing tutorial*. Guru99. https://www.guru99.com/software-testing.html

Hamilton, T. (2023, November 25). *What is Data Driven Testing? Learn to create Framework*. Guru99. https://www.guru99.com/data-driven-testing.html

Homès, B. (2012). *Fundamentals of Software Testing*. https://download.e-bookshelf.de/download/0000/7533/44/L-G-0000753344-0002285971.pdf

Hooda, I., & Chhillar, R. S. (2015, February). *Software Test Process, Testing Types and Techniques*. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0fbe1b5515e747025d950658fbc039e98b29b801

Holling, D., Hofbauer, A., Pretschner, A., & Gemmar, M. (2016). Profiting from Unit Tests for Integration Testing. 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). https://doi.org/10.1109/icst.2016.28

How to get height and width of element in selenium WebDriver. (n.d.). https://www.software-testing-tutorials-automation.com/2015/01/how-to-get-height-and-width-of-element.html

Ibrahim, J., Hanif, S., Shafiq, S., & Faroom, S. (2019, July). *Emerging Trends in Software Testing Tools & Methodologies: A Review*. ResearchGate.

https://www.researchgate.net/profile/Saeed-Faroom/publication/338160048_Emerging_Trends_in_Software_Testing_Tools_Methodologies_A_Review/links/5e033efa92851c83649937f6/Emerging-Trends-in-Software-Testing-Tools-Methodologies-A-Review.pdf

K, D. R. (2016, October 30). Test Data in automation framework. Selenium Easy. https://www.seleniumeasy.com/selenium-tutorials/test-data-in-automation-framework

Lazić, L. (2010). Software testing optimization by advanced quantitative defect management. Computer Science and Information Systems, 7(3), 459–487. https://doi.org/10.2298/csis0909230081

Lucian. (2019, October 25). Test Analysis and Test Design are key for software testing. Cania Consulting. https://cania-consulting.com/2019/10/25/a-test-manager-guide-to-test-analysis-and-test-design/

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing*. Wiley.

Nair, V. (2022, January 5). Why is Test Coverage an Important Part of Software Testing? *Medium*. https://vipinrajnair.medium.com/why-is-test-coverage-an-important-part-of-software-testing-f7ac168a6d90

Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann.

Oracle. (n.d.). What is an exception? https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Orakzai, W. by: R. U. (2023, August 7). Software testing: Defect, bug, error, and failure. Baeldung on Computer Science. https://www.baeldung.com/cs/software-testing-defect-bug-error-and-failure

Performance testing. (2021, December 7). Selenium. Retrieved December 3, 2023, from https://www.selenium.dev/documentation/test_practices/discouraged/performance_testing/

Pressman, R. S. (2001). Software Engineering. https://repository.dinus.ac.id/docs/ajar/Software_Engineering_-_Pressman.pdf

Pressman, R. P., S. (2001). Software Engineering: A Practitioner's Approach (Fifth Edition). Thomas Casson.

Nidhra, S. (2012a). Black Box and White box testing Techniques - A literature review. International Journal of Embedded Systems and Applications, 2(2), 29–50. https://doi.org/10.5121/ijesa.2012.2204

R Indira. (2001). Test Result Reporting. Test Result Reporting.

Release Testing Explained: Best Practices and Examples. (n.d.). LambdaTest. Retrieved December 5, 2023, from https://www.lambdatest.com/learning-hub/reliability-testing

Santi, P. a. D. A., Afwani, R., Albar, M. A., Anjarwani, S. E., & Mardiansyah, A. Z. (2022). Black Box Testing with Equivalence Partitioning and Boundary Value Analysis Methods (Study Case: Academic Information System of Mataram University). In Advances in Computer Science Research (pp. 207–219). https://doi.org/10.2991/978-94-6463-084-8_19

Sawant, K., Tiwari, R., Vyas, S., Sharma, P., Anand, A., & Soni, S. (2021). Implementation of Selenium Automation & Report Generation Using Selenium Web Driver & ATF. 2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT). https://doi.org/10.1109/icaect49130.2021.9392455

Security testing and selenium. (2021, February 16). https://www.seleniumtests.com/2021/02/security-testing-and-selenium.html

Selenium IDE · Open source record and playback test automation for the web. (n.d.). https://www.selenium.dev/selenium-ide/

Sharma, M., & Chandra, B. S. (2010). Automatic Generation of Test Suites from Decision Table - Theory and Implementation. 2010 Fifth International Conference on Software Engineering Advances. https://doi.org/10.1109/icsea.2010.78

Shuaibu, I., Machina, M. M., & Ibrahim, M. (2019). Investigation onto the Software Testing Techniques and Tools: An Evaluation and Comparative Analysis. International Journal of Computer Applications, 177(23), 24–30. https://doi.org/10.5120/ijca2019919685

Singh, S. (2023, April 23). How to Automate TestNG in selenium ? | BrowserStack. BrowserStack. https://www.browserstack.com/guide/testng-framework-with-selenium-automation#:~:text=and%20parametrization%20features.-,What%20is%20TestNG%20in%20Selenium%3F,tests%20more%20efficiently%20and%20effectively.

Softwaretestinghelp (June 2023) Cause And Effect Graph – Dynamic Test Case Writing Technique For Maximum Coverage With Fewer Test Cases.
https://www.softwaretestinghelp.com/cause-and-effect-graph-test-case-writing-technique/

Singh, S. (2013). Analysis of Bug Tracking Tools. International Journal of Scientific & Engineering Research, 4(7).

Sneha, K., & Gowda, M. (2017). *Research on Software Testing Techniques and Software Automation Testing Tools*. https://sci-hub.ru/10.1109/icecds.2017.8389562

Study of various bug tracking tools. (2020). LC International Journal of STEM, 1(2). https://doi.org/10.47150/jstem027

Testing Guru. (2020, January 29), What is the cost of defects in software testing? https://www.testing.guru/what-is-the-cost-of-defects-in-software-testing/The Selenium Browser Automation Project. (n.d.). Selenium. https://www.selenium.dev/documentation/

Tiwari, G. (2023, February 17). How to perform UI Testing with Selenium | BrowserStack. BrowserStack. https://www.browserstack.com/guide/ui-testing-with-selenium Test data for Selenium - guides. (n.d.). https://guides.flood.io/scripting-and-tools/selenium/test-data-for-selenium

Toledo, F. (2018, November 20). *Increase test coverage with automation*. dzone.com. https://dzone.com/articles/increase-test-coverage-with-automation

TSaar et al., (2016). Cross-Browser Testing in Browserbite. Springer Link https://link.springer.com/chapter/10.1007/978-3-319-08245-5_37

Tufano, M. (2020, September 11). Unit Test Case Generation with Transformers and Focal Context. arXiv.org. https://arxiv.org/abs/2009.05617

Types of Testing. (2023, October 28). Selenium. https://www.selenium.dev/documentation/test_practices/testing_types/

Umar, M. A., & Zhanfang, C. (2019b). A Study of Automated Software Testing: Automation Tools and Frameworks. *International Journal of Computer Science Engineering (IJCSE)*, *8*(6), 217–225.

Unadkat, J. (2023, February 6). *6 Best Practices in Selenium Automation*. BrowserStack. https://www.browserstack.com/guide/best-practices-in-selenium-automation

Unadkat, J. (2023, February 6). Exceptions in selenium Webdriver : How to handle it | BrowserStack. BrowserStack. https://www.browserstack.com/guide/exceptions-in-selenium-webdriver

Unadkat, J. (2023, February 3). What is Selenium RC : Difference from Webdriver | BrowserStack. BrowserStack. https://www.browserstack.com/guide/selenium-rc-tutorial#:~:text=Selenium%20RC%20is%20an%20important, applications%20against%20any%20HTTP%20website.9

Vaidya, N. (2023, February 2). How to Read/Write Excel Data using Apache POI Selenium | BrowserStack. BrowserStack. https://www.browserstack.com/guide/read-data-from-excel-using-selenium

Vijay. (2023, June 25). *Usability testing tutorial: A complete getting started guide*. Software Testing Help. https://www.softwaretestinghelp.com/usability-testing-guide/

Warne, M. (2022, December 5). Automation testing selenium - a step by step guide for Beginners. koenig. https://www.koenig-solutions.com/automation-testing-selenium-a-step-by-step-guide-for-beginners#:~:text=Steps%20for%20automation%20testing%20using%20Selenium%20WebDriver&text=Step%202%3A%20Configure%20Eclipse%20and,configure%20WebDriver%20with%20Eclipse%20IDE.&text=Step%206%3A%20Create%20a%20test%20script%20and%20select%20%27run%27.

What is feature testing and why is it important. (2023, June 30). Software Testing Help. https://www.softwaretestinghelp.com/feature-testing-tutorial/

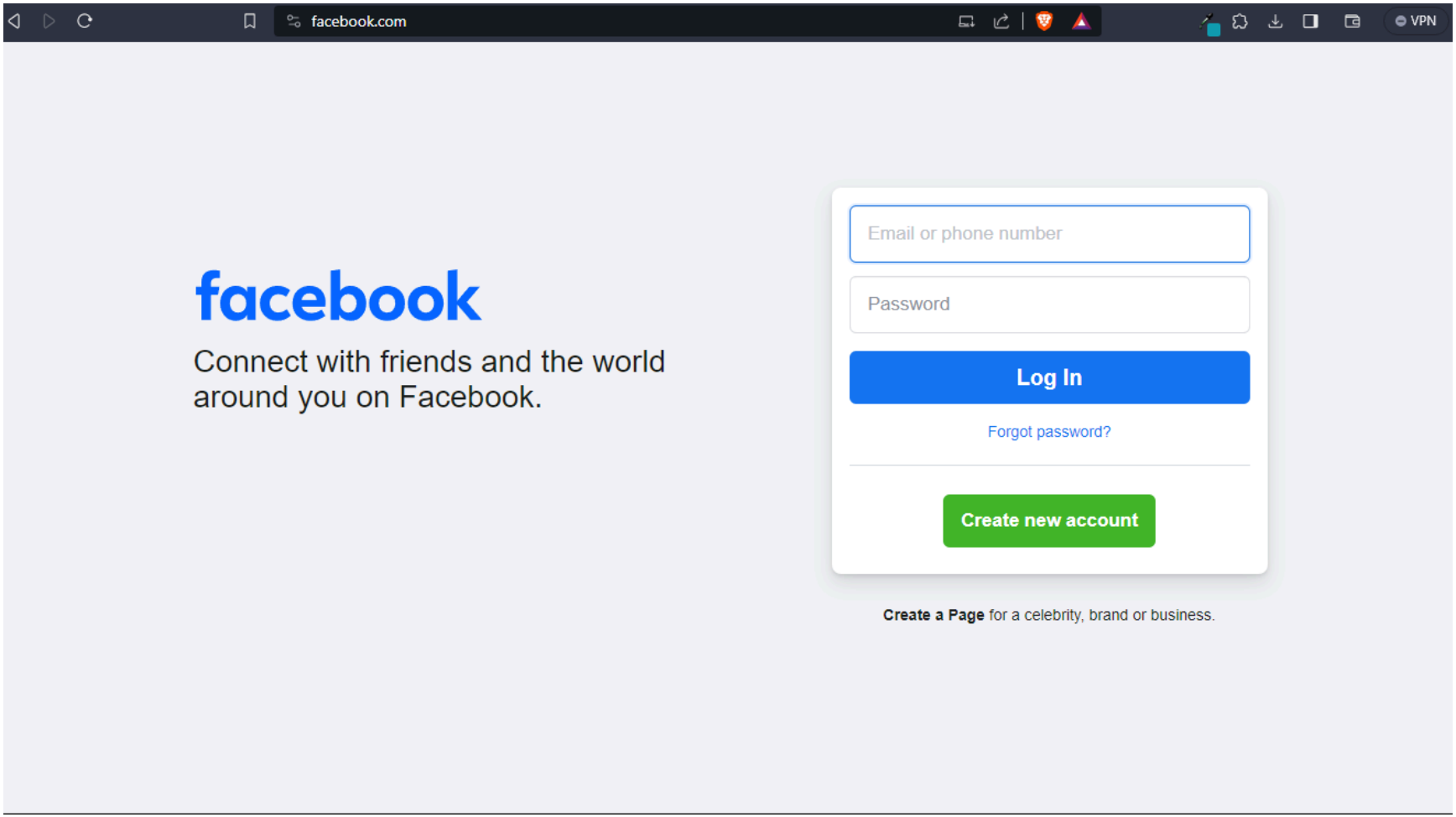
Weyuker, E. J., & Vokolos, F. I. (2000, December). *Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study*. IEEE. https://sci-hub.ru/https://ieeexplore.ieee.org/abstract/document/888628

Zhang, T., Chen, J., Jiang, H., Wang, L., & Xia, X. (2017). Bug Report Enrichment with Application of Automated Fixer Recommendation. *2017 IEEE 25th International Conference on Program Comprehension*. https://doi.org/10.1109/icpc.2017.28

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., & Weiß, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering*, *36*(5), 618–643. https://doi.org/10.1109/tse.2010.63

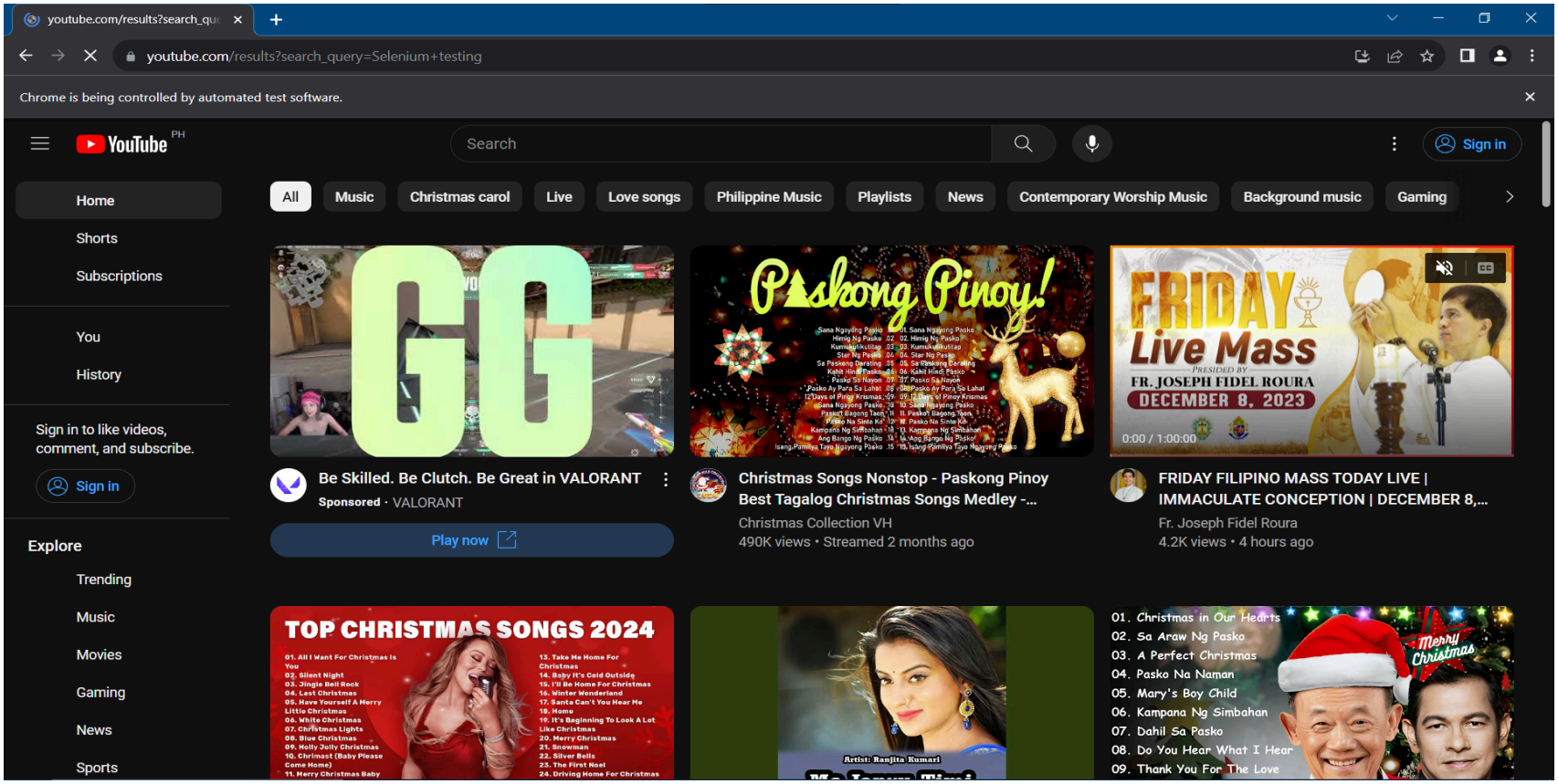
Screen Shots

Unit and integration testing with Cross-feature testing

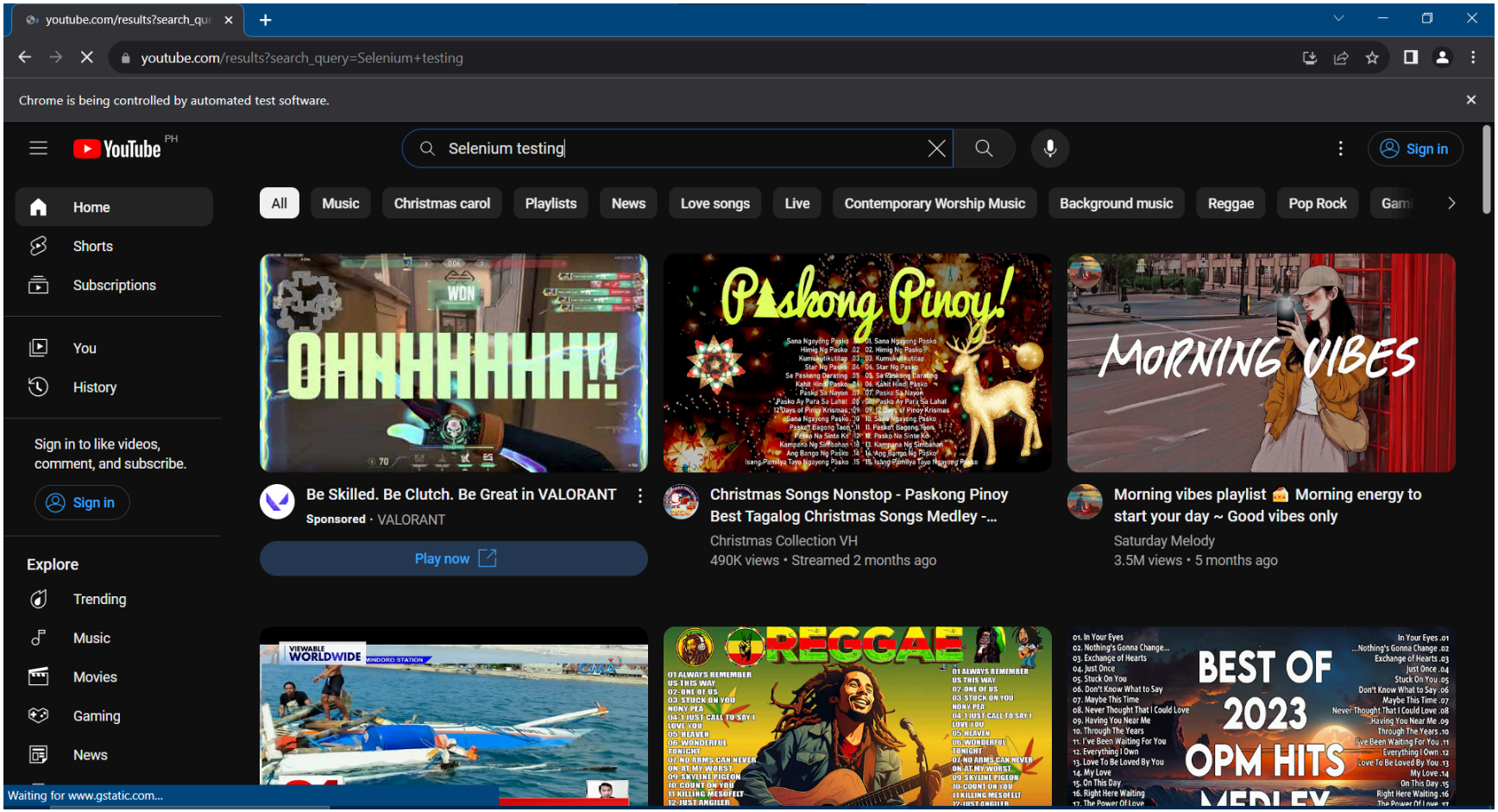


Site used for Unit and Integration testing with Cross-Feature Testing

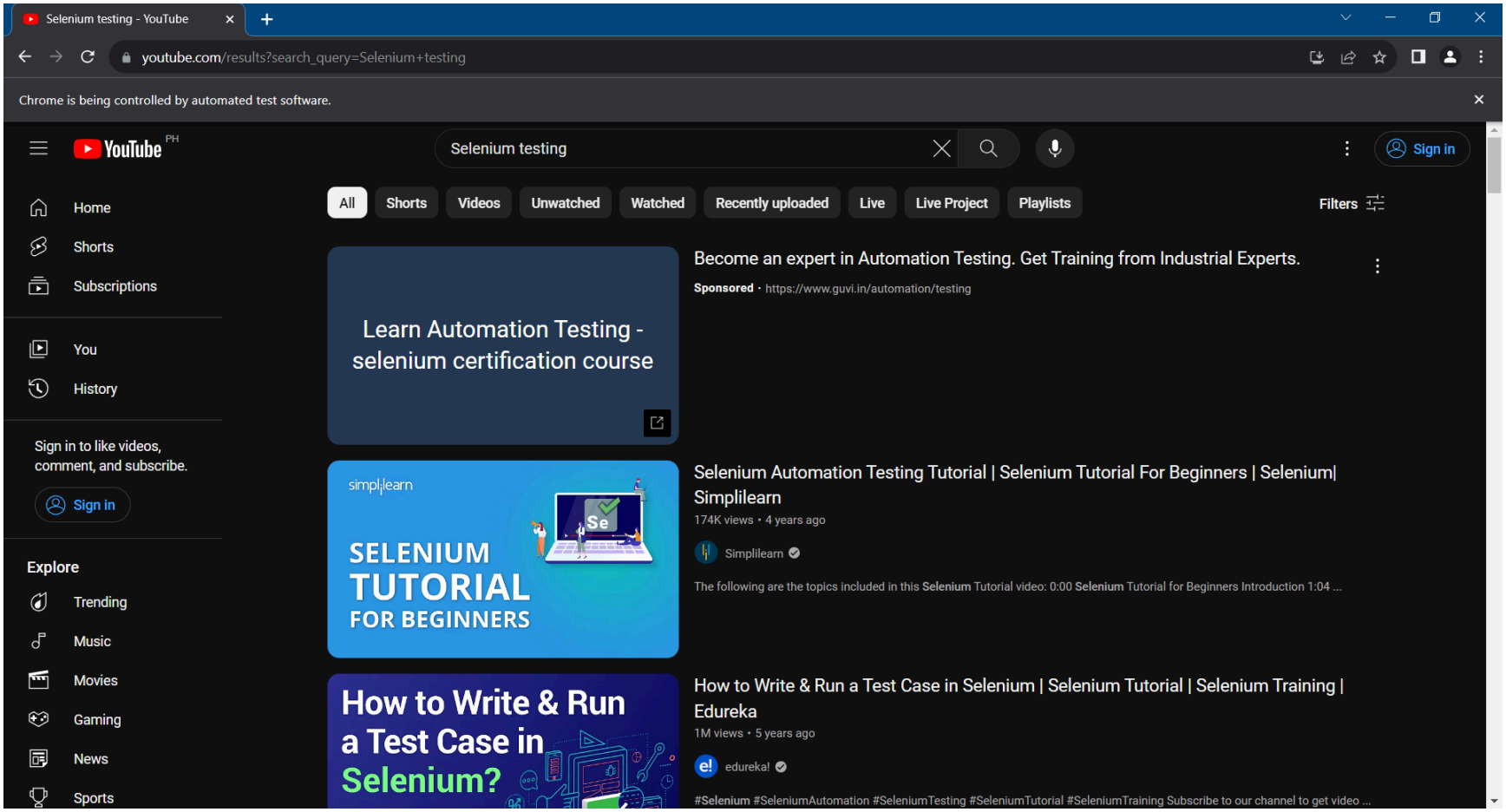
Automate ‘Search Video’ feature of free video sharing website, ‘Youtube’ with Selenium Webdriver.



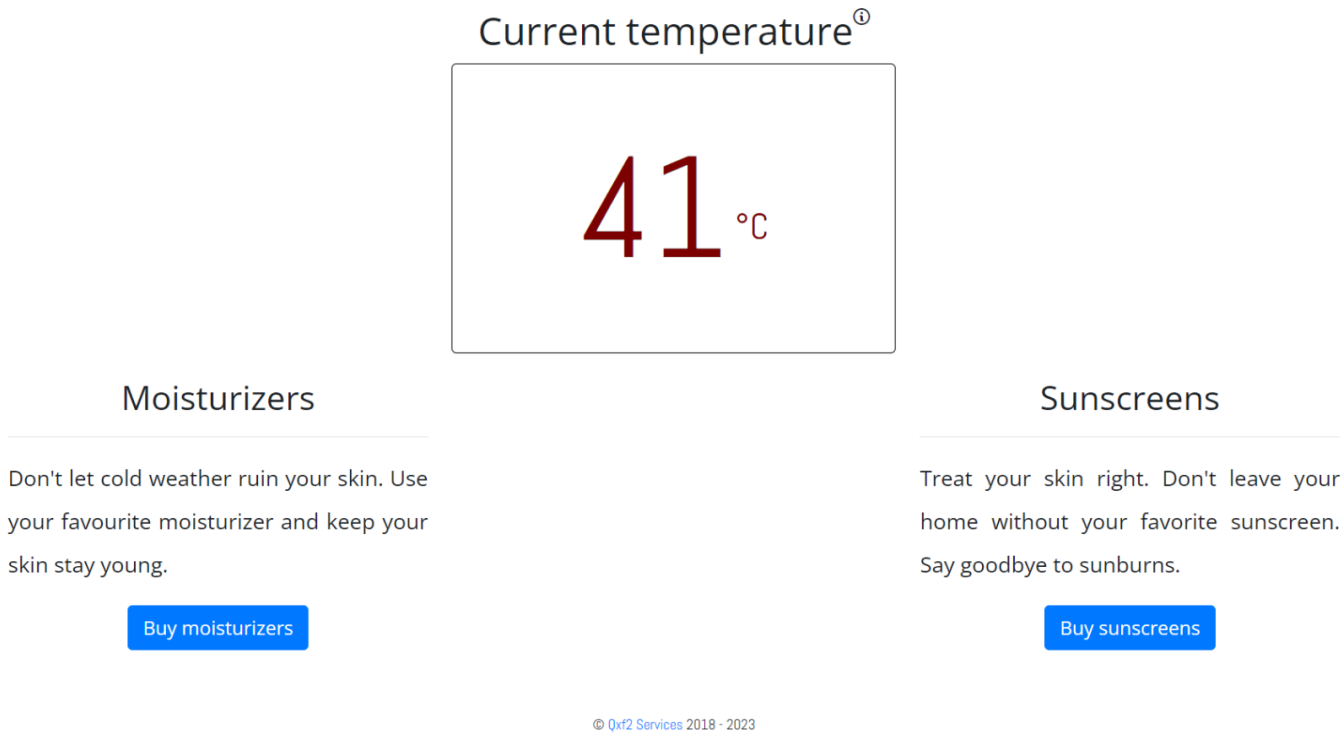
Open/Access Youtube Website



Search Video Name



Video Search Result will be displayed



Site used for Functional testing

Test Case Examples

Unit and integration testing as well as Cross feature testing.

Test Case Scenario No.1:

Automate 'Login' feature of 'Facebook' with Selenium WebDriver integrated with Junit. And Cross-feature Testing for different browsers of the 'Login' feature of Facebook with Selenium Webdriver

```
package seleniumWebDriver;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.edge.EdgeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;
import java.util.Arrays;
import java.util.Collection;

@RunWith(Parameterized.class)
public class AutomatedLogin {

    private WebDriver driver;
    private String browser;

    @Before
```

```
    public void setUp() {
        // Set up WebDriver based on the specified browser
        switch (browser.toLowerCase()) {
            case "chrome":
                System.setProperty("webdriver.chrome.driver",
"C:\\Users\\bruce\\Downloads\\selenium-first\\selenium\\chromedriver.exe");
                driver = new ChromeDriver();
                break;
            case "firefox":
                System.setProperty("webdriver.gecko.driver",
"C:\\Users\\bruce\\Downloads\\selenium-first\\selenium\\geckodriver.exe");
                driver = new FirefoxDriver();
                break;
            case "edge":
                System.setProperty("webdriver.edge.driver",
"C:\\Users\\bruce\\Downloads\\selenium-first\\selenium\\msedgedriver.exe");
                driver = new EdgeDriver();
                break;
            default:
                throw new IllegalArgumentException("Unsupported browser: " + browser);
        }

        driver.get("https://www.facebook.com/login/");
    }

    @After
    public void tearDown() {
        // Close the browser after all tests are executed
            driver.quit();
    }

    @Parameterized.Parameters(name = "{index}: Test with browser={0}, email={1}, password={2}")
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][]{
                {"chrome", "youremail@gmail.com", "yourPassword"},
```

```
            {"firefox", "invalid_email", "invalid_password"},
            {"edge", "jillahmarie@gmail.com", "1"},
            {"firefox", "", "12345678901234567890123456789012345678901234567890123456789012345678901234567890"}
    });
}

private static String email;
private static String password;

private String emailParam;
private String passwordParam;

public AutomatedLogin(String browser, String emailParam, String passwordParam) {
    this.browser = browser;
    this.emailParam = emailParam;
    this.passwordParam = passwordParam;
}

private void setCredentials(String email, String password) {
    AutomatedLogin.email = email;
    AutomatedLogin.password = password;
}

@Test
public void testLogin() {
    try {
        setCredentials(emailParam, passwordParam);
        login();
        // Additional assertions based on the specific scenario can be added here
    } catch (Exception e) {
        e.printStackTrace();
        // Handle the exception appropriately (e.g., log the error, take a screenshot, etc.)
    }
}
```

```
    private void login() {
        // Wait for the email input field to be present
        waitForElementPresent(By.id("email"));

        driver.findElement(By.id("email")).sendKeys(email);
        driver.findElement(By.id("pass")).sendKeys(password);
        driver.findElement(By.id("loginbutton")).click();
    }

    private void waitForElementPresent(By locator) {
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        wait.until(ExpectedConditions.presenceOfElementLocated(locator));
    }
}
```

Smoke testing

Test Case Scenario No.2: Automate 'Search Video' feature of free video sharing website, 'Youtube' with Selenium WebDriver.

Automation Test:

TEST ID.	TEST SCENARIO	DESCRIPTION	TEST STEP	EXPECTED RESULT	ACTUAL RESULT	STATUS
0	Youtube Smoke Testing: Automating the 'Search' functionality of Youtube.	This smoke tests aims to verify the 'Search' functionality of Youtube using Selenium WebDriver.	1. Open the Chrome browser and Navigate to Youtube. 2. Access Search Bar and type the desired video to watch. 3. Submit the search. 4. Wait for the Search Results	• Youtube website is accessible. • The search box is working. • The result of the search are displayed. • The first video in the	• Youtube website is accessible. • The search box is working. • The result of the search are displayed. • The first video in the	Pass

			5. Verify the Search Results. Verifying that the searched video is displaying after search. 6. Close the Browser.	website is printed in the console for verification that the search is working.	website is printed in the console for verification that the search is working.	

Test Case No.2: Smoke Testing using Youtube 'Search' functionality

```
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;
import java.time.Duration;



public class YouTubeSmokeTest {



   public static void main(String[] args) {
```

```
// Set the path to your ChromeDriver executable
System.setProperty("webdriver.chrome.driver", "chromedriver.exe");



// Access YouTube
WebDriver driver = new ChromeDriver();
driver.get("https://www.youtube.com/");



WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement searchBox = wait.until(ExpectedConditions.elementToBeClickable(By.name("search_query")));



// Search for a video
searchBox.sendKeys("Selenium testing");
searchBox.submit();



// Wait for search results to load
wait.until(ExpectedConditions.presenceOfElementLocated(By.id("video-title")));



// Print the title of the first video
```

```
        WebElement resultElement = driver.findElement(By.id("video-title"));
        String videoTitle = resultElement.getText();
        System.out.println("First video title: " + videoTitle);



        // Perform additional verifications for smoke testing
        verifySearchResults(driver);



        // Close the browser
        driver.quit();



        System.out.println("YouTube Smoke Test performed successfully!");
    }



    private static void verifySearchResults(WebDriver driver) {
        // Verify that search results are displayed
        WebElement resultsContainer = driver.findElement(By.id("contents"));
        assert resultsContainer.isDisplayed() : "Search results are not displayed";
    }
}
```

Functional testing

Test Case No.3: Functional Testing using Weathershopper's 'Add Item'

```
// Generated by Selenium IDE
import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.core.IsNot.not;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.Dimension;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.JavascriptExecutor;
```

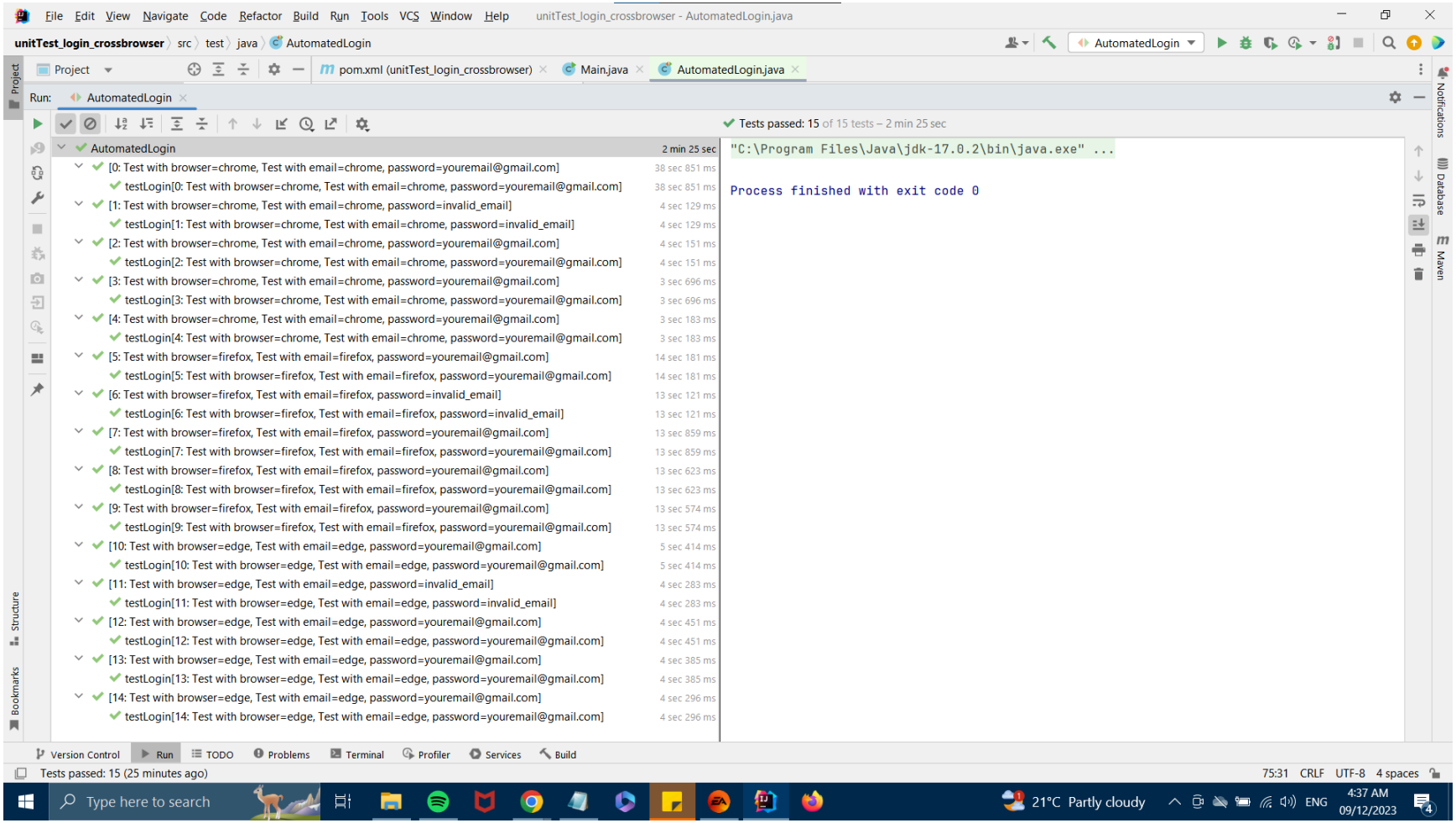
```
import org.openqa.selenium.Alert;
import org.openqa.selenium.Keys;
import java.util.*;
import java.net.MalformedURLException;
import java.net.URL;

public class AddItemCartTest {
  private WebDriver driver;
  private Map<String, Object> vars;
  JavascriptExecutor js;
  @Before
  public void setUp() {
    driver = new ChromeDriver();
    js = (JavascriptExecutor) driver;
    vars = new HashMap<String, Object>();
  }
  @After
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void addItemCart() {
    driver.get("https://weathershopper.pythonanywhere.com/");
```

```
    driver.manage().window().setSize(new Dimension(782, 920));
    driver.findElement(By.cssSelector(".octicon")).click();
    driver.findElement(By.cssSelector(".text-center:nth-child(1) .btn")).click();
    driver.findElement(By.cssSelector(".row:nth-child(2) > .text-center:nth-child(1) > .btn")).click();
    driver.findElement(By.id("cart")).click();
    driver.findElement(By.cssSelector("td:nth-child(1)")).click();
    driver.close();
  }
}
```

Test Case: Adding item to cart

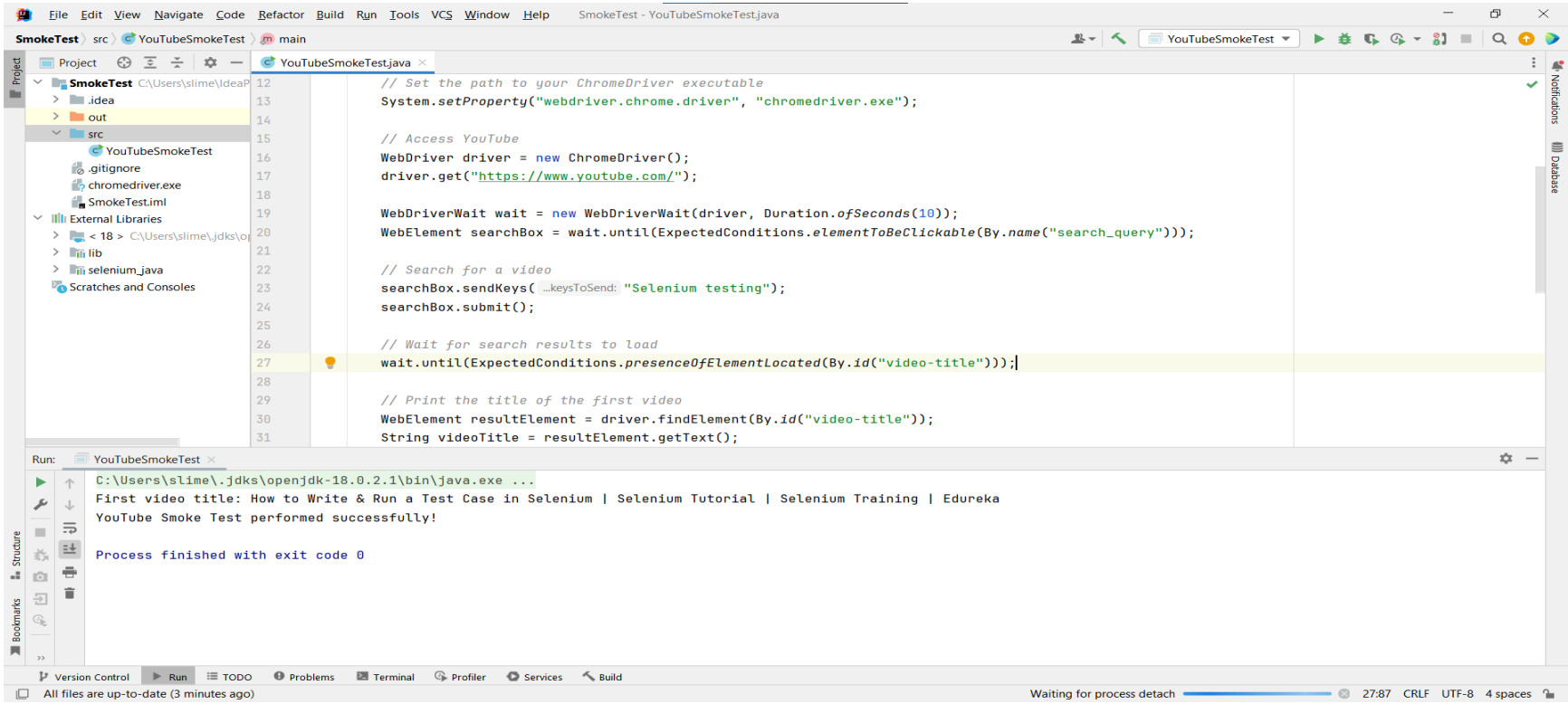
Testing Result

Test Case No.1: Unit and integration testing as well as Cross feature testing with Regression testing using Facebook Login



Test Case: Automate 'Login' feature of 'Facebook' with Selenium WebDriver integrated with Junit. And Cross-feature Testing for different browsers of the 'Login' feature of Facebook with Selenium Webdriver.

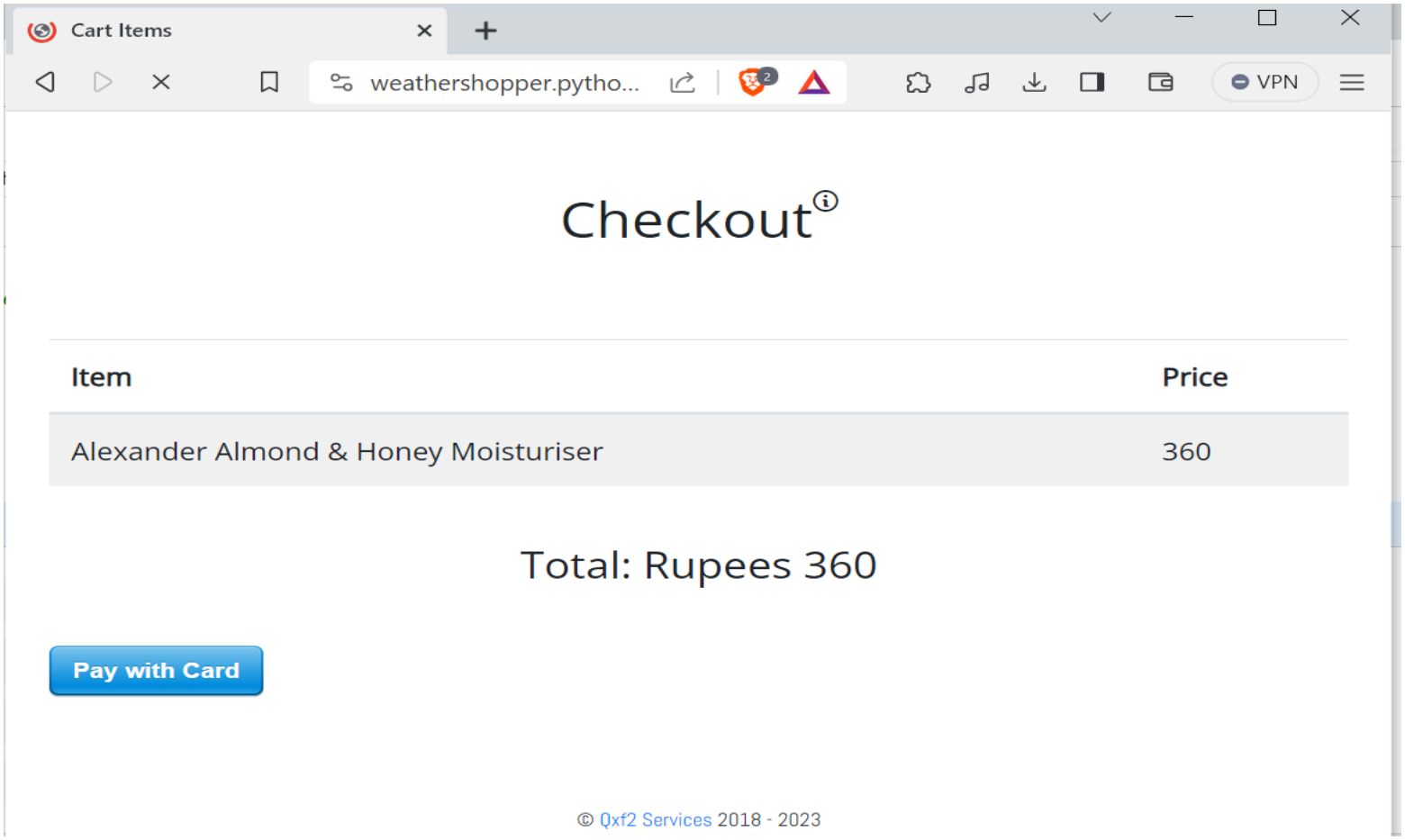
Test Case No.2: Smoke Testing using Youtube 'Search' functionality



Test Case: Automate 'Search Video' feature of free video sharing website, 'Youtube' with Selenium WebDriver.

(Verifying that the searched video is displaying after search)

Test Case No.1: Functional Testing using ‘Add Item’ functionality of Weathershopper webiste



Test Case: Verifying that the item is added to the cart

Supporting Documentation/Visuals

Unit and integration testing as well as Cross feature testing

Action	Test Data
Open Facebook	N/A
Login using Valid Credentials	“youremail@gmail.com”, “your password”
Login using Invalid Credentials	“invalid_email”, “your password”
Login with Minimum password length	“youremail@gmail.com”, “1”
Login with Maximum password length	“youremail@gmail.com”, “12345678901234567890123456789012345678901234567890123456789012345678901234567890”
Login with Missing password	“youremail@gmail.com”, “ ”

Test data in the website automating the ‘Login’ feature of Facebook

Action	Test Data
Open Youtube Website	N/A
Search for a video	“Selenium Testing”
Wait for search results to load	N/A
Print the title of the first video	N/A
Additional Verifications	N/A
Close the browser	N/A

Test data in the website automating the ‘Search’ feature of Youtube

Action	Test Data
Find the search box element	N/A
Input the search query “Selenium Testing”	“Selenium Testing”
Submit the search form	N/A

Test data in the program for automating the ‘Search’ feature of Youtube

Action	Test Data
Open ‘https://weathershopper.pythonanywhere.com/’ Website	N/A
Click button “Buy Moisturizers”	N/A
Click button “Add”	N/A
Check the cart for the added moisturizer	N/A
Close the browser	N/A

Test data in the program for automating the ‘Add Item’ feature of Weathershopper