TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

Azad Husen 184631IASM

MAINTAINABLE TEST SUITE DESIGN USING PAGE OBJECT MODEL IN SELENIUM WEBDRIVER

Master's Thesis

Supervisor
Vladimir Viies
PhD

Tallinn 2020

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Azad Husen .....
(signature)

Date: 18th May, 2020

Abstract

The aim of this Master's thesis is to design a test suite using page object model in Selenium Webdriver and measure how page object model is more maintainable than conventional test suite design. There will be analysis of updating test suite if an UI changes, the items needs to update if locator id changes and number of lines of code to change requires in future. Two test suites will be developed with and without page object model. Although the page object model is an established model in test automation, the goal of the thesis is to prove how maintainability can be achieved by using it with a real life example. The methods used in this Master's thesis will be tested for *https://igavesti-ou.myshopify.com/* as an example.

The issues that need to be resolved for design and development of maintainable test suites: Choosing a test automation tool, define how to proceed on by choosing page objects, proving maintainability, optimize results and other simulated scenarios.

This Master's thesis strives to give an analysed overview of test suite maintainability and tries giving answers to the questions that may appear.

The thesis is in English and contains 73 pages of text, 6 chapters, 18 figures, 12 tables.

List of abbreviations and terms

API	Application Programming Interface
AUT	Application Under Test
CD	Continuous Deployment
CI	Continuous Integration
COM	Component Object Model
CPU	Central Processing Unit
DOM	Document Object Model
GUI	Graphical User Interface
HP	Hewlett-Packard
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IE	Internet Explorer
IT	Information Technology
OCP	Open Closed Principle
OLE	Object Linking and Embedding
OS	Operating System
PIP	Python Package Installer
POM	Page Object Model
QA	Quality Assurance
QTP	Quick Test Professional
RoR	Ruby on Rails
SDLC	Software Development Life Cycle
SQL	Structured Query Language
SRP	Single Responsibility Principle
UFT	Unified Functional Testing
UI	User Interface
WATIR	Web Application Testing in Ruby
XML	Extensible Markup Language

Table of Contents

List of Figures

List of Tables

1. Introduction

Software testing is one of the most important and crucial phases in the software development life cycle process,consuming an average of 40 to 70 percent of the time in the development process[1]. In the agile software development, testing is fundamentally important as it enables visibility and enhances communication and feedback to developers [2]. Test automation is mandatory for the success of web applications in the long run: it saves a lot of time in testing and helps to release web applications with fewer defects[3]. Automated testing of web applications reduces the effort needed in manual testing, but it can't replace manual testing. Automated GUI regression testing tools are very popular for test automation in IT industry. According to the popular mythology, people with less programming language experience can use those tools and create extensive test suites. However, maintenance of these test suites becomes costly in case of anything changes in future in that project. As a result, designing of maintainable test suites are the main concern in test automation for the success of the test automation in a software project.

Test suite is a collection of test cases which can be used for test execution. This is a common term in both manual and automation testing. As the test suit consists hundreds or thousands of test cases, a lot of test cases are correlated in a way that can minimize redundancy. This is where maintainability of test suite concern. In the design of test suite, if you don't consider maintainability of test suite, then keeping track of hundreds of test cases really becomes cumbersome.

Selenium is one of the most popular tools for web application automation across different browsers and platforms. It is being used in the software domain not only for test automation but also for automating administration tasks as well. Mainly, it consists of two main parts Selenium IDE and Selenium Webdriver. Selenium IDE; a Chrome and Firefox browser plugin where anyone can record and playback interactions of the browser during a bug generation and verify it by adding assertions which can be used as an automated test suite. On the other hand, Selenium Webdriver; a collection of language specific bindings to drive a browser - the way it is meant to be driven. Selenium Webdriver has bindings for many programming language so test cases can be written in pure programming language using your own favourite tools[4]. It is best when when you want to create robust, browser-based test automation suites, scale and distribute scripts across many platforms. In Selenium

Webdriver, test suites are implemented by using a programming language which requires time and effort. This is why, maintainability is the first priority during the design phase of the test suite.

Page Object Model(POM) is a popular design pattern in different test automation frameworks. The basis of POM is to model the web pages as objects, applying the same programming language used to write the test cases[3]. As a result, functionalities of an web page considered as services (i.e: methods) derived from page objects which can be called in any test cases of that project. This eliminated repetitive codes as well as it becomes easier to change code in case of any changes comes in the project.

This thesis is aimed at exploring and designing maintainable test suites using Selenium Webdriver. The main focus would be showing how POM helps to design a maintainable test suite. Along with design, maintainability parameters in a test suite developed with and without POM will be analyzed. I will use a real life e-commerce website to implement test cases for the test suites. This could be useful for manual testers who wants to switch to automation or the new engineers who wants to start career in test automation. This could be an industrial level use case for any e-commerce platform.

1.1 Background

Test automation is nothing new, its been around for 40+ years. However, extensive automation testing started for enterprise applications from the beginning of this century. Automated functional testing is based on testing web page by creating test cases that automate the interaction with a web page and its elements. Test cases are used to fill-in and submit forms or click on hyperlinks automatically. Test cases can be programmed using general programming languages, but choosing a programming language must consider several aspects of the application under test. It is better to use a programming language which provides specific library with user friendly APIs. Built in methods can be used in order to command e.g., click a button, fill a field and submit a form. Finally, test cases are completed with assertions to validate [5].

Page Object Model is already an established model in the market, but I have chosen to proceed on in this thesis because e-commerce automation could be tricky as this deals with payment processing and user data. On the other hand, the same code can be used for automating mobile version of the web page. Also, many published papers for POM is in Java, so this was one of challenges to analyse maintainability of test suites in POM using Selenium Webdriver in Python.

1.2 Problem

Who wrote this piece of code?? I can't work like this!! —Any programmer

This is a common problem in software development when we work in a team; the same happens in test automation as well. Today almost all of the studies related to automation testing are exclusively based in using Java programming language. However, Python is a popular programming in software development, so this could be powerful in test automation as well.

Here is the basic paradigm for GUI-based automated regression testing: [6]

- Design a test case, then run it.
- If the program fails the test, write a bug report. Start over after the bug is fixed.
- If the program passes the test, automate it. Rerun the test (either from a script or with the aid of a capture utility). Capture the screen output at the end of the test. Save the test case and the output.
- Next time, run the test case and compare its output to the saved output. If the outputs match, the program passes the test.

First problem is this is not cheap, second problem is this approach increases additional cost, third problem is these tests are not powerful, and fourth problem is in practice, many test groups automate only the easy-to-run tests.

In addition to that, most of the companies are using paid automation tools, or they are building their framework. Apart from this, the main problem was to provide a solution in low budget for a client, but the requirements are to develop maintainable test suites which can be easily readable, changeable without having strong coding experience. For the paid tools; initial cost is lower than free tools, but it becomes costly in the long run due to monthly subscription[7].

The purpose of the given thesis is to show how to design and develop a maintainable test suite in Python. The test suite will be designed in a way that anyone can understand the logical flows if they have knowledge about the AUT. So, the client can understand all the test cases and run regression test as per their need. Also, they can change locator id if it changes later. In addition to that, they can use the same test suites if the language changes. Finally, they can hire a test automation engineer to update the test suites and it will be less time-consuming for an engineer having this test suite in hand.

The goal of this thesis is to improve the readability of tests and therefore also the maintainability of the test suite. In order to prove this, the goal is to quantify the effort needed to realign the two test suites using the following metrics: the time required to update test suite if UI changes, if locator id changes, if the language of the product changes. The properties of its source code[8] determine the maintainability of a software system. Maintenance is something we have to face after delivery of the code, two main goals of this thesis are:

1. When the program's UI changes, what amount of work do we need to do to update the test suite with the goal that they precisely reflect and test the program?
2. When the UI language changes, (for example, English to French), how hard is it to modify the suite so they precisely reflect and test the program?

Maintainability is a vast topic that can be applied in any kind of development. For example, house development should be designed in a way that maintenance can be done easily on periodical basis. Software maintenance is not about fixing wear and tear. Software is not physical, and therefore it does not degrade by itself the way physical things do [8]. Test automation is also software development but it does not deliver direct product to the customers. This is why it was not a concern during the implementation of automation tools, but from the last 7-8 years POM becomes popular in test automation. A paper titled 'Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study' showed how to improve maintainability in a test suites where test suite developed using Java in Selenium. In related to that, the author published several papers related to test automation in the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. However, Selenium IDE lost the popularity by these years and now Webdriver is the main popular feature of Selenium. A TalTech student did bachelor thesis titled 'Selenium-based web Test Automation Framework Development' where he implemented a test automation framework. Though he used page object pattern, but his focus was not to show how to design in a way to make maintainable test suite. The current state of the art is that POM is already an established model in the test automation but my thesis would show how to design and implement maintainable test suites.

2. Testing of web applications

In today's aggressive and rapidly changing world of software development, it has become critical for the organization to test their web application before releasing to the market. By performing different kinds of testing, organizations can gain confidence that their product will work without problems or difficulties for the end-users. In testing, companies may follow multiple strategies. Still, the common focus of the trial is to make sure the intended functionality is working as expected and find the bugs in the early stage of development.

Based on these criteria, there are mainly two types of testing: Functional and Non-Functional testing. Testing which checks the business logic of the product is functional testing, any other kinds of testing termed as non-functional.

2.1 Definition of Testing

Testing is defined as a process of evaluation that either the specific system meets its originally specified requirements or not [9]. In other words, testing is a series of actions or steps to verify the intended result is achieved as per the initial requirements. In simple words, software testing is an activity to check whether the actual results match the expected results and to ensure that the software system is defect free [1].Testing approach depends on the software development life cycle model, but it should provide the exact knowledge about the quality of the product to the stakeholders.

Testing can be considered as a risk based activity where a tester must understand how to minimize a large number of test sets into small manageable test cases and cluster what is important to test in a particular feature or product release. Testing cost and bugs finding have a relationship where number of missed bugs becomes less when amount of testing increases.

According to Figure 1, Where the graph of testing and number of missing bugs intersect is determined as the optimal point of testing. The effective testing goal is to do that optimal amount of tests so that extra testing effort can be minimised [9].

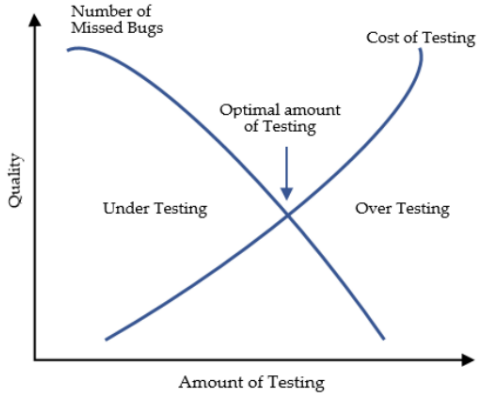

Figure 1. *Every Software Project has optimal test effort[9]*

2.2 Types of Testing

Testing is a very wide topic in the area of IT systems where it includes various methods, types, and levels or stages of testing. Attributes of testing depend on what application is being tested, how big is the feature, and who does the testing. Also, the tester knowledge about the AUT plays a vital role in order to divide testing activities into several groups of types. Static testing and dynamic testing are the base in order to differentiate types of testing. Static testing is an approach where code is not executed; it manually checks the code, project requirement documents, and design documents to find errors. On the other hand, code must be executed in order to do dynamic testing to check the functional behavior of a software product. In high level, the following categories can be derived in order to classify types of testing:

2.2.1 According to the Way of Testing

Testing can be done by a human or by the computer. When a tester manually executes a particular set of tests, we call it manual testing. When the same set of tests performed by a computer, we call automated testing. In reality, test automation cannot be a replacement for manual testing, but it helps the tester to execute repetitive testing over a more extended period. Moreover, some tests are time-consuming to execute manually; for example, if you want to automate thousands of actions in a specific time, sometimes it's not possible to do by the human at all.

Manual testing involves manual tasks like setting up test environment, execute the test tasks and report the found bugs, review the results. This process can be done by following a test plan or by exploratory testing[10].

Following a test plan is the formal process of test steps needs to execute in order to check functionality of the application under test. According to IEEE 29119-3 [11], it should contain at least a test plan identifier, an introduction, test items, the features to be tested, the features not to be tested, an approach, item pass/fail criteria, stakeholder information, testing communication and a schedule. Because it deals with predefined steps, it may be executed by a less skilled tester. On the other hand, it requires time and effort from an expert tester or test lead to document and maintain test plans.

Exploratory testing known as ad-hoc testing where a tester explore the product and test. Most of the cases, its an unplanned testing but this approach gives opportunity to think out of the box. As this testing does not require test plan or a minimum test plan is sufficient, it helps to get a result quickly. Creativity and the experience of the tester is the key in order to discover important bugs quickly.

Automated testing known as test automation involves test execution without human interference. In this testing, tester or developer write the scripts to run the test using another software or tools. Basically, the same test plan can be used to develop automated tests. Automated tests execution includes, however, other prerequisites [10]:

- Ability to run a subset of all tests.
- Automatic set-up and record environmental variables.
- Running the test cases.
- Capturing the results.
- Comparing actual and expected results and highlighting the differences.
- Analysing the results and processing them in a comprehensive and clear way.

Both manual and automated testing have advantages as well as disadvantages. The circumstances of the application under test helps deciding the right way of testing. The circumstances includes how big is the application, how big is the testing budget, and what is the deadline to finish the testing. Manual testing takes less time to setup initially but automated testing benefits in the long run. However, testing speed is slow in manual whereas automated testing is faster and uninterrupted so human resource can dedicate their time to something else. Overall, test automation could take time and investment on initial setup but it can reduce costs for regression testing in bigger projects [10]. Comparison between manual and automation testing can be seen in Table 1.

Table 1. *Comparison of the Manual and Automation Testing*

No.	Manual Testing	Automation Testing
1	Time consuming as the test is executed by human and sometimes tedious.	Test execution by software tools, so it is significantly faster than human and boring is not an issue for machine.
2	Less reliable due to human error.	More reliable as it is performed by tools/scripts.
3	Investment required for human resources.	Investment required for tools.
4	Manual testing is practical when test run is schedule once or twice (i.e: frequent repetition is not required).	Automation testing is practical when a repetitive test execution required over a long time.
5	Manual testing allows human observation and exploratory testing which helps to find UX and usability issues	Automation testing does not allow human observation; so less chance to guarantee UX and usability issues.

2.2.2 According to Depth

The next chosen attribute for the types of testing is according to the depth of tester insight into the AUT. This approach is known as box approach, where tester knowledge about the internal structure of the product is essential when designing test cases. This testing methods are traditionally divided into white and black-box testing[1], but there is another called grey-box testing, which has mixed features of white and black-box testing.

White- box testing; According to the name, this is a testing approach where a tester can see inside the box. Clear-box testing and denotes technique, when the tester knows the underlying implementation of an application, so he/she can decide how to develop the tests according to this knowledge[10]. In this testing approach, the tester should understand the structure of code to test. As a result, this kind of testing usually done by a designated tester or developer.

Black-box testing; Likewise white-box testing, the name of this testing approach, describes the type where a tester cannot see inside the box. Black box testing treats the software as a "black box". It indicates the way of testing, where the tester does not know the details of how the AUT works[10]. An output of input is essential, but not how the action is done. This is why this type of testing can be done by end-users as well as tester or developer.

Grey-box testing; This type of testing known as semi-transparent testing; includes having knowledge about the internal structures of AUT for the test planning, but testing at the end-user or black-box level. An example of this is testing of functionality of web applications because to perform this type of testing, the knowledge of web page structure is required, and simultaneously it is not important how the functionality testing performed[10]. Usually, this type of testing performed by end-users, testers, and developers.

2.2.3 According to Scope

The last but the most critical attribute of testing is the scope. The scope denotes the size of the tested parts of the AUT, or more specifically, what portion of the product needs to be tested to achieve a result. The scope of the testing needs to consider where the application stands in the SDLC pyramid, and when the tests are written for it. The motivation behind the testing pieces of software comes in play because testing by parts can quickly identify the error in small portion testing in isolation. Moreover, this type of testing helps a tester to find bugs in the early stage of development. Testing by parts/units gives an overview of small features before integrating into the full product. As a result, testing according scope helps to identify the faulty feature in the early stage of development and debugging in order to release top quality product.

Unit testing; A unit is the smallest piece of software code that can be tested by a mechanism. Mostly this type of testing is done by the developer or a white-box tester who can compile the code. It is required to break the software development into a set of units/-modules where each module assigned to a different team or different individual. After the completion of each module or unit, it is tested by the developer just to check whether the developed module is working by the expectation or not; this is termed as unit testing [9]

Integration testing; The next step after unit testing is the integration of the units/modules. A single unit may work alone as the unit test passed, but it is not obvious that it will work when modules are connected to each other. As a result, once the modules of a single software system have been developed independently, they are integrated, and often errors arise in the build once the integration has been done [9]. For example, testing the interaction with the database or payment processor requires integration testing.

System testing; The final testing step in the SDLC is system testing, which is testing the whole software from every perspectives[9]. Several units aggregated into one to make a component; several components integrated into one to create a system (i.e., software product). System testing validates the full and completely integrated software product.

These are the main criteria for the classification of testing, but there are many other scenario's can be considered to divide testing. The following diagram can describe classifications of testing:

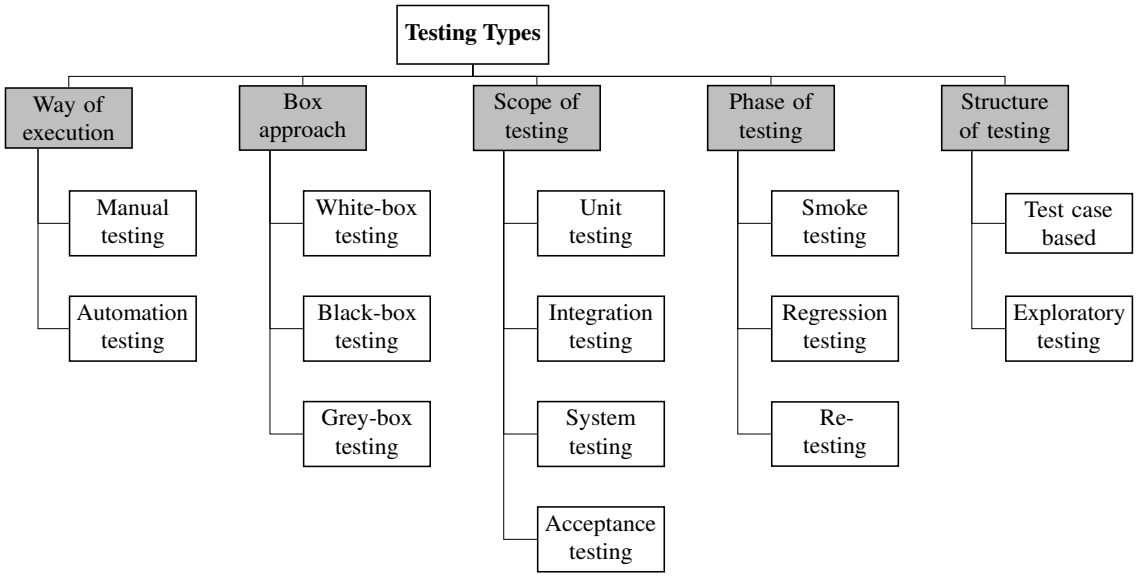

Figure 2. *High level testing types based on various methods, types and scope or stages of the testing*

2.3 Continuous Integration and deployment

"Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove."— Martin Fowler, Chief Scientist, ThoughtWorks

In software engineering, continuous integration (CI) is the practice of merging all developers' working copies to a shared mainline several times a day[12]. It is a software development practice where everyone on the engineering team is continuously integrating small code changes back into the codebase. After each change that they're making, there's a suite of tests that runs automatically that checks the code for any bugs or errors or anything like that. A basic chart below describes how this happens practically:

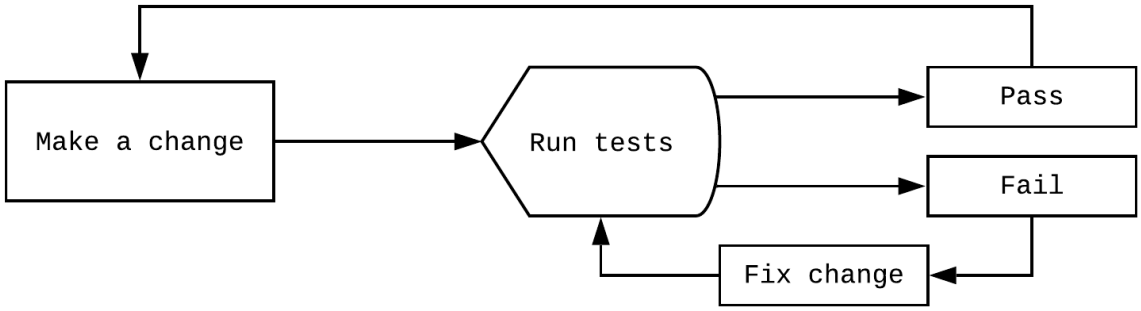

Figure 3. *Continuous Integration in practical life*

In the Figure 3, Step one is when a developer change code; they make a future branch, push it up, submit a pull request. Then the integration tool runs tests automatically, and that's pretty important before the next move. The tests need to be run in a consistent way so that everyone on the team has full confidence that they're running the tests in the same style, and they can confidently work on the same version of code. If the test pass, the current developer is good to go and can keep making changes. The same developer can go right back around, submit another pull request. If the tests fail, he/she need to make sure to fix the code. Fix the code until tests pass and then continue developing. So it's this beautiful cycle. The idea behind this that the development team wants to catch these bugs as soon as they can. So they're not lingering over a long period of time.

The software quality is improved by minimizing the integration risks. This is the risks of testing product at the end of development life cycle instead of testing simultaneously during the development to catch the bugs and fix as soon as possible. As a result, integration testing is feasible not only for integration but also for the unit and system testing[10].

Continuous deployment is the next step after CI; it is the ongoing delivery process of features which are updated, tested, and ready for release. CD is part and parcel in the agile development world. Business matters on continuous delivery where organizations are in pressure for quick delivery and shorter time to get a return of investment. The continuous deployment process flow can be illustrated below:

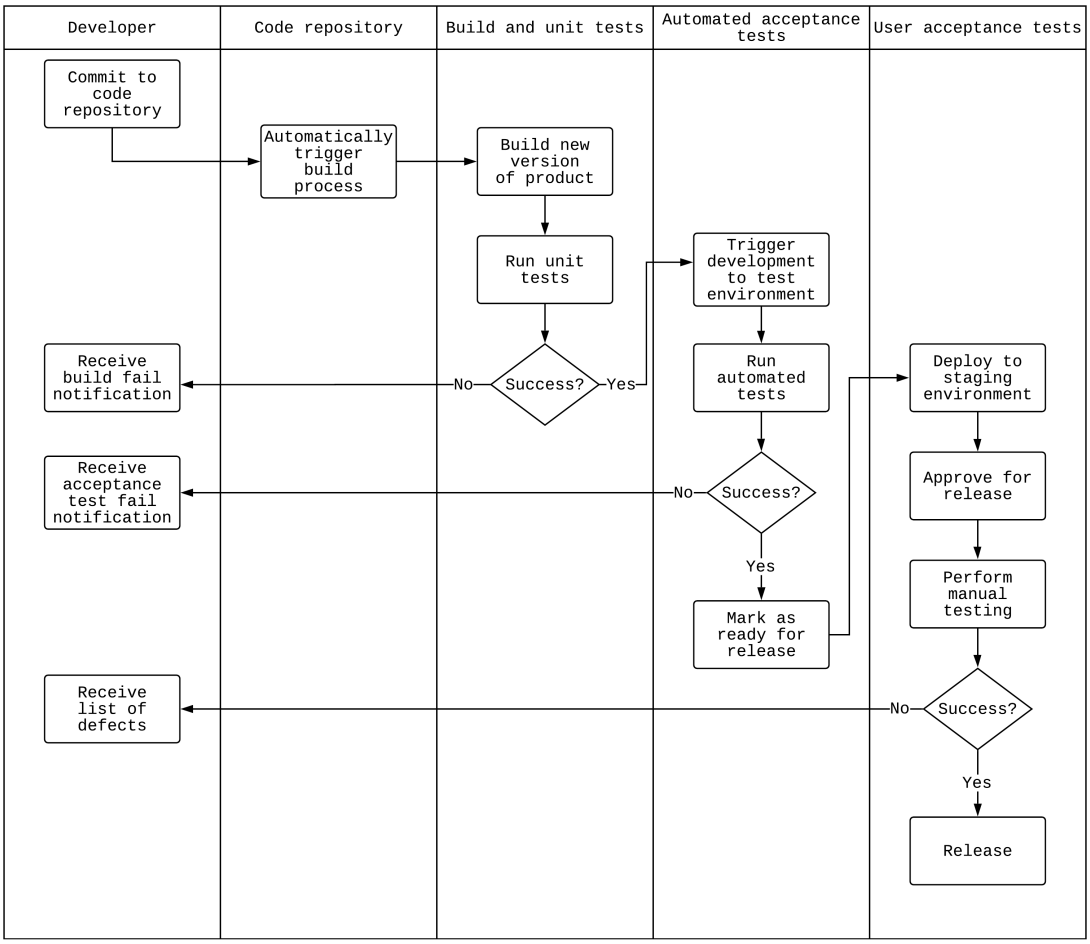

Figure 4. *Continuous Deployment process flow [13]*

According to Figure 4, CD is a long but automated process where a developer commits source code for a software program to the code repository. There are several code repository tools available in the market. Once code is stored in an organized way, code repository triggers the build process, and then it automatically runs the unit tests. If the unit test passed, the build goes to automated acceptance tests where testing happens in the test environment. This is a system-level testing where it runs fully automated tests, and if it becomes a success, the build is marked as ready for release. Most of the companies follow user acceptance tests after an automated acceptance test. In this step, ready for release build deployed to the staging environment and run another automated test to get approved for release. Finally, QA tester performs manual testing, and the product gets released. In any of the steps, if the automated or manual testing failed and the release blocker arises,

it goes back to the developer to hot-fix and commit the code change. Again, the cycle of the CD starts. For the management of this process, there are several available tools in the market; Gitlab is one of those.

Key principles of CI

To achieve highest possible software quality, CI process has to dully follow key principles listed below:

- Software code should be maintained in the code repository.
- Build and tests of the software project should be automated.
- Code changes should be committed to the code base every day.
- Each delivering of the code should trigger project build and testing process on dedicated machine.
- The building and testing should be as fast as possible.
- User acceptance test should be run in the production environment.
- Results should be clearly visible and authors of the code delivery should be notified that their code delivery might cause failure.
- The latest version of application should be easily accessible, and this process of releasing the application should be automated [10][14].

Benefits and Problems of CI

The advantages of CI are much more valuable than the disadvantages, but those problems cannot be ignored. Here are two major benefits of CI:

- A major benefit of CI is reduced risk, which is the main challenge of any project. Because of CI, there is no long integration, so you know where you are, what works and what is not working; outstanding defects in your project.

- The second benefit of CI makes a solution to frequent deployment. As a result, your user gets new features more quickly, and interactive feedback comes from the user side as well. This process brakes the ice between the customer and the development team.

On the other hand, a couple of main drawbacks of CI are below:

- Initial investments into hardware, that is usually a dedicated machine with CI tool running on it and its slaves on which the actual CI build and testing is performed. The cost could be partially limited by the virtualization of some of the machines.

Virtualization is a creation of a virtual hardware platform, or operating system, etc. Also, Tests need to be automatized, which can extensively increase the overall development expenses.

- Security vulnerabilities need to be taken into account because an attack to the CI system can mean disclosure of confidential information or system shutdown, which can have a negative impact on application development process [10]

2.4 Test Automation Approaches

Test automation is not just for test case execution [15]. According to the author of this article [15]; To work more efficiently and effectively, test engineers must be aware of various automated testing strategies and tools that assist test activities other than just test execution. However, automation does not come for free, so it must be carefully implemented. Test automation strategy comes from the thinking of aftereffect of test suit implementation, which is test maintenance. In agile software development, continuous integration plays a vital role where software QA engineers should think about how to integrate automated testing during development, deployment, and delivery, beyond mere test execution. However, software engineers will be able to get the highest benefit from such automation only if they know available strategies and tools for test automation.

To determine strategies, anyone must be aware of the building block of a testing framework. Based on the testing types and key principles of CI, a basic building block of test automation framework can be sketched below:

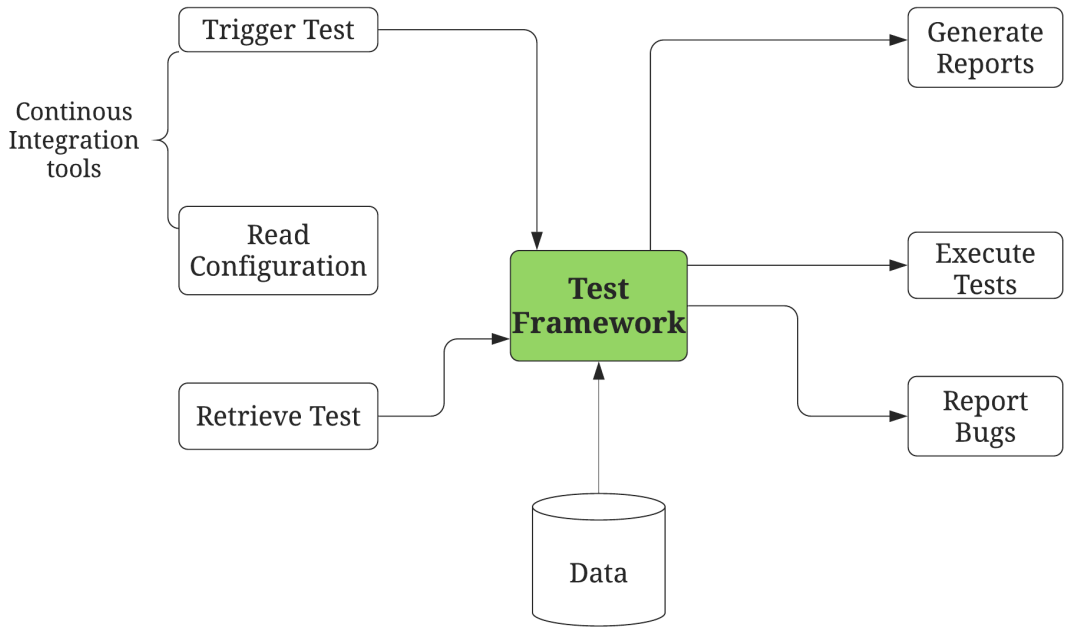

Figure 5. *Basic building block of a test automation framework*

In the Figure 5, We can divide inputs and outputs by the left and right-hand parts. The test framework depends on several external sources where to inject from the CI/CD tool requires to get trigger test. The framework must be able to read configurations and web element object repository. The significant advantage of object repository is the segregation of locators from the test case; more details are in 4. In addition to that, the framework will need to read data from a data source like excel files. On the right side, during or after test execution, bug reporting should be done in a professional way, including all details. Such details are screenshot, video, logs, etc. which helps on debugging by the developer. Test execution reports should be generated and send the report to stakeholders, usually HTML formatted reports.

Test automation is software development, so the approaches are similar to the SDLC. Even if we use a simple record/playback tools, some sort of codes are generated in the background. Like any project management, test automation strategy requires planning, analysis, design, implementation, and maintenance.

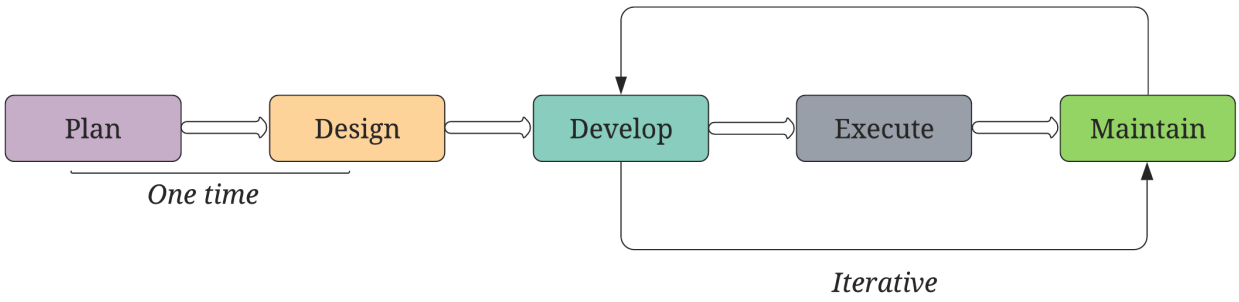

Figure 6. *Test automation approaches aligning with SDLC*

According to Figure 6, Test automation planning and designing are one time approach, whereas development, execution, and maintenance is an iterative approach. In the planning phase, feasibility study must be done in order to shortlist the relevant test cases for automation and choosing the right test tool as per client requirement. Then, selecting a test automation framework is a vital part of the design phase. There are multiple options to proceed with. For example, linear test automation framework, data-driven framework, key-word driven framework, etc. Test data management plans and techniques for choosing reusable libraries are also included in this step. After that, the heart of test automation starts, which is the development of the test automation script. In this period, creating test data and building execution flow started, which are the input of the next stage. Finally, test execution starts where creating a test environment, deploying test suites, test script execution to reporting bugs are the integral parts. Last but not least, the maintenance of test suites, update assets, archiving test scripts, and using the test automation for the regression run must be planned as well. However, based on the scenario of the product update, these strategies can be changed in the test automation process.

To better understand how this process works, the author of this paper 'Test Automation Not Just for Test Execution' derived six testing activities with an enormous potential for automation[15]. These could be termed as test automation life cycle:

1. Test-case design.
2. Test scripting.
3. Test execution.
4. Test evaluation.
5. Test results reporting.

6. Test management and other test engineering activities.

Test-case design; This activity is the first duty of a test engineer when they plan to move into test automation. Not all test cases designed for the manual test can be automated. The test case design outputs a suite of test cases (input and expected output values) or test requirements (for example, control flow paths to cover) [15]. The output of this activity is being used by the next activity called test scripting.

Test scripting; Test scripting outputs either manual test scripts (in a variety of formats) or automated test suites for use in test execution. Testers have been manually performing test scripting for many years [15].

Test execution; Test execution runs the test cases on the AUT and records the results or observes the AUT's output or behavior. Decisions made during the previous activities affect test execution. For example, if a tester develops all the tests as automated test suites, test execution will obviously be fully automated. In contrast, if the test team develops all its tests as manual test scripts, test execution must be manual. Test execution is partially automated if some scripts are automated and others are manual [15].

Test evaluation; There are usually three approaches for evaluating the test outcome (pass or fail):

- A human tester makes the judgment.
- The developers incorporate (hard-code) test evaluations as verification points (assertions) in the test code.
- The developers build "intelligent" (learning) test oracles, using machine learning and AI.

Test results reporting

This is usually the last phase; it reports test verdicts and defects to the developers for fixing - for example, through defect-tracking systems[15].

Test management and other test engineering activities; These activities include test set minimization (which includes test redundancy detection), regression test selection, and test repair (conducting maintenance on broken automated tests when the AUT has changed)[15].

3. Evaluation of the Tools

The test automation approach discussed in the previous chapter 2, first step is planning where you must choose the right test automation tool as per the client requirement. Companies who have less budget are not interested in test automation, but due to the complexity of software development, its not easy to ignore. Though they have a financial limitation, the aspiration to have automation testing leads them to choose the right tool from available options. An automated test is more effective when time, cost, and usability are concerned [16].

There is a wide verity of test automation tools that focuses on automating web and mobile applications, either open source or commercial. While those tools that support a wide range of applications, with better features and functionality, may require additional costs[16]. In this paper, I tried to automate an e-commerce based web applications for a startup company; I concentrated mainly on open source tools which are suitable for enterprise usage. The initial requirement is to automate their checkout process for different browsers, especially for the web, then enhance for mobile browsers. That means cross-browser support must be there where it has to support all major browsers including Firefox, Internet Explorer, Safari, and Google Chrome, on the major Operating Systems (Linux, Windows, Mac). The next requirement was to provide a programmatic way of creating test scenarios from manual test scenarios; CI integration is the next target. Finally, the tool has to be enabled for testing rich web applications (i.e: Ajax-based web sites).

3.1 Criteria for choosing the tools

The criteria for choosing a test automation tool may differ from organization to organization. This is because of the initial investment in purchasing tools, the high initial cost in designing test cases, and may require additional training of human resources. To get the maximum benefit of test automation, criteria for choosing the right tool must be set in a way to make test automation more reliable, programmable, reusable, comprehensive, and maintainable in the long run. In addition to that, a tool which can support more excellent test coverage should be in the list of evaluation parameter.

To analyze the features of automated testing tools, we need to identify the features to be used for distinguishing similarities and dissimilarities of each tool. According to Mittal [17], when selecting the best tool among automated testing tools, we can consider these key points: Support to platforms and technology, flexibility for testers of all skill levels, feature-rich but easy to create automated tests that are reusable, maintainable and resistant to changes in the applications user interface.

Table 2. *Evaluation criteria of test automation tools[16][17][18]*

Criteria	**Definition**
Cost	Whether free or licensed
Cross platform	How many operating systems supported
Cross browser	How many browsers supported
Record playback	Ability of tool to record scripts
Script language	Programming languages used to edit test scripts or for the creation of testing scripts
Ease of learning	How easy the tool is used
Available resources	How easily the resources found in online
Programming skills	Programming skills needed or not
Data driven	The ability of tool to reduce efforts like making it possible to make the scripts access the different sets of input data from external source like data tables, excel sheets
Report generation	How result is represented
Training cost	The training cost for the tool if exist where low is less than 500, medium is 500-1500 USD, high is 1500++

3.2 Comparison of tools

Several researchers compared the automation tools based on the above-mentioned criteria. In the article titled 'Comparative review of the features of automated software testing tools' the author concluded that if the project cost is to be given higher consideration, open-source tools such as Selenium is the better option. If the availability of support, ease of learning, report generation are to be considered, licensed tools such as QTP/UFT is a good option [16]. In another article titled 'Comparative Analysis of Automated Software Testing Tools' the author analyzed only commercial tools and concluded that Loadrunner is the best tool[18]. They could be correct in their point of view, but Loadrunner is best for load testing where client-server application, network, and web performance needs to be tested. Authors of the article [19] 'A Critical Analysis of Software Testing Tools' could

not come to a conclusion but their observation looks practical where for a particular testing purpose, tradeoffs can be made to select the best tool depending on the size of the project, the budgeted cost for testing, the platform of the application and also the language that is used to develop the project. According to my chosen parameter from the client requirement, I found Gamido [16] has the latest analysis where they showed the comparative review of the selected automated software testing tools based from the evaluation parameter used in the above section:

Table 3. *Comparative review of automated software testing tools[16][9][20]*

Criteria	**Selenium**	**Watir**	**Test Complete**	**QTP**	**Ranorex**	**Load Runner**
Cost	Open source	Open source	Licensed	Licensed	Licensed	Licensed
Cross platform	Windows Linux Mac	Windows Mac Linux	Windows	Windows	Windows	Windows Linux Mac
Cross browser	Chrome, Firefox, IE, Opera, Safari	Chrome, Firefox, Opera, IE,Safari	Chrome, Firefox, Opera, IE	Chrome, Firefox, IE	Chrome, Firefox, Opera, IE, Netscape, Safari	Any browser
Record & playback	Support	Support	Support	Support	Support	Support
Script language	Java, Ruby, Python, PHP, C#	Ruby, Java, C#	Vbscript, C#, Javascript, C++, Delphi	Vbscript, Java, C#, Delphi	Vbscript, C++, C#, Python	Vbscript, C, Vb, C#, Javascript
Ease of learning	Experience needed	Easy to learn	Experience needed	Easy to learn	Easy to learn	Experience needed
Available resources	Plenty	Limited	Limited	Limited	Limited	Limited

Continues...

Table 3 – *Continues...*

Criteria	**Selenium**	**Watir**	**Test Complete**	**QTP**	**Ranorex**	**Load Runner**
Programming skills	Needs to have programming skills	Partial	Needs to have programming skills	Partial	Partial	Partial
Data driven	Yes	Yes	Yes	Yes	Yes	Yes
Training cost	Low	Medium	High	High	High	High
Report generation	Html	Html, Xml	Html, Xml	Html, Xml -gives executive summary of test, gives statistics in the form of pie charts	Html -with executive summary, with graphs for faster and better comparison of defects in every run	Does not provide graphical representation of results

3.2.1 Selenium

Selenium is one of the efficient test automation tools become popular among QA engineers because of the nice test automation framework and flexibility of coding in almost all popular programming languages. Jason Huggins originally developed Selenium in 2004 as an internal tool at ThoughtWorks, which is a privately owned global software company [21]. Mainly Selenium used for web application test automation, but it's not limited to test automation only. Web scraping is another usage of Selenium where anyone can scrape data from javascript generated content from a webpage. Selenium comes with three variants:

- Selenium IDE: A Chrome and Firefox add-on that will do simple record-and-

playback of interactions with the browser[22].

- Selenium Webdriver: A robust driver that controls the browser's behavior. This is a solution when you want to create browser-based regression automation suites and tests, scale and distribute scripts across many environments. There are a lot of different implementations to support main modern browsers(Firefox, Chrome, Internet Explorer, Opera, Safari) and even some headless browsers(HTMLUnitDriver and PhantomJS). [4]
- Selenium Grid: It is a server for evaluating instances of web browsers operating on remote machines. If you want to scale up by spreading and running tests on multiple machines and control different environments from a central point, making testing easy to run against a large combination of browsers/OS, then Selenium Grid is the option[22].

3.2.2 Watir

Watir (Web Application Testing in Ruby) is pronounced like water. It is an open source family that uses ruby libraries to automate web browsers. Watir enables testers to write tests that are easy to read and maintain. Watir is simple and flexible[1]. Most important variations of Watir are:

- Watir-classic: Watir-classic makes use of the fact that Ruby has built-in object linking and embedding (OLE) capabilities. As such it is possible to drive internet explorer programmatically. Watir-classic operates differently than HTTP based test tools, which operate by simulating a browser. Instead, Watir-classic directly drives the browser through the OLE protocol, which is implemented over the Component Object Model (COM) architecture [23].
- Watir-webdriver: Watir-webdriver is a modern version of the Watir API based on Selenium; Jari Bakken has implemented the Watir API as a wrapper around the Selenium 2.0 API in Ruby [23].
- Watirspec: Watirspec is an executable specification of the Watir API like RubySpec is for Ruby [23].

3.2.3 Test Complete

TestComplete is a functional test automation tool developed by SmartBear Software. TestComplete gives testers the ability to create automated tests for Windows, Web, Android, and iOS applications. Tests can be recorded, scripted or manually created with keyword-driven operations and used for automatic playback and error logging[24]. This is an automated UI testing tool with artificial intelligence.

3.2.4 QTP

Quick Test Pro (QTP) is a GUI based test automation tool for recording and playback, a part of the HP quality center tool suite. It was originally written by Mercury Interactive, which was acquired by HP (Hewlett Packard) in 2006 [1]. It is a tool used to automate functional and regression tests for various software applications and environments. HP's Quick Test Professional uses the VBScript scripting language to specify the test procedures and to manipulate the objects and controls of the test AUT. QTP also enables us to test Java applets and applications, and multimedia objects on Applications as well as a standard Windows application, Java, Visual Basic applications and .NET framework applications.This works by defining and executing the necessary operations (such as mouse clicks and keyboard events) of the application user interface or a webpage. Although HP's Quick Test Professional is usually used for "UI Based" test case automation, some "Non-UI Based" test cases, such as file system operation and database testing, can also be automated. [18].

3.2.5 Ranorex

Ranorex GmbH is a German software development company developed the GUI test automation tool called Ranorex Studio. This framework is used for desktop, cloud, and mobile apps quality testing. Using common programming languages such as C and VB.NET Ranorex Studio supports the development of automated test suits. The main features of this tool are:

- Recognition of GUI object, filtering of GUI elements using RanoreXPath proprietary technology
- Object-based recording and replaying using Ranorex Recorder which records the user's desktop or web-based interactions and generates users-maintainable scripts that can be edited with the Ranorex studio action editor. Record and replay for

actions such as key presses and touch actions is supported on mobile devices. The recorded actions are available as both C and VB.NET code [25]

3.2.6 Load Runner

HP's Load Runner is a test automation tool from Hewlett-Packard for load testing: system behavior and performance are examined while a real-time load is generated. It works by creating virtual users that replace current users and generate loads. The load means to make thousands of simultaneous operators to put the application through the severities of actual user load, although gathering information from key setup mechanisms. [18].The key components of LoadRunner are[26]:

- Load Generator generates the load against the application by defined scripts
- VuGen (Virtual User Generator) for generating and editing scripts
- Controller controls, launches, and sequences instances of Load Generator - specifying which script to use, for how long, etc. During runs, the Controller receives real-time monitoring data and displays status.
- Agent process manages the connection between Controller and Load Generator instances.
- Analysis assembles logs from various load generators and formats reports for visualization of run result data and monitoring data.

4. UI Automation- Page Object Model and other design patterns

Writing test script using Python is not a tough job as its easy to code and easy to read. It requires finding elements and perform systematic actions in a webpage. Consider the below example of a simple selenium script that will navigate to Taltech website and identifies the search field and enters the query and click on the search button. I used the Taltech website just for an example other than the AUT I will be using in my real experiment.

```
from selenium import webdriver

class searchTaltech:
   driver =webdriver.Chrome("../drivers/chromedriver")
   driver.get("https://taltech.ee/en/")
   driver.find_element_by_id('search-text').send_keys('Computer
                                        Systems')
   driver.find_element_by_id('search-text').submit()
   assert 'No Results' not in driver.page_source
   driver.quit()
```

In the above code, a test created for UI based web applications using Selenium libraries have two parts:

- Located the UI elements by locators.
- Performed actions on these elements.

From the code, it looks like maintaining is very easy because of fewer lines of code. However, when you need to test all the features, it becomes complex UI with many pages and elements. Then there will many test cases which will increase lines of code, and the maintenance of the code will be difficult sometimes un-maintainable. In addition to that, if the same element (i.e., search field) is being used in 15 test cases, then the id of that field needs to be changed in 15 places, which is time-consuming and impractical when test engineers continue to add and extend tests.

4.1 Page Object Model

To solve the problem mentioned above, an approach of a formalized design pattern called Page Object Model (POM) incubated in the software development industry. The POM design pattern principle is about the separation between test classes and pages (business objects) where it allows test projects to decouple responsibilities (tests VS. page logic) and expand codes in the test script. This is a design model that distinguishes the UI elements and tests or operations conducted on them. Usually, the UI components are implemented as Page Objects with all the associated logic. Tests are entirely independent to execute these Page Object operations. Thus, features provided by a web page become "services" provided by the specific page object (i.e., methods) that can easily be called in any test case. Therefore, all the web page specifics are within the object of the website. Adopting the page object pattern allows the test developer to operate at a higher abstraction level (clearly, unless page objects are required)[3].

The diagram below shows the sequence between test and a page object:

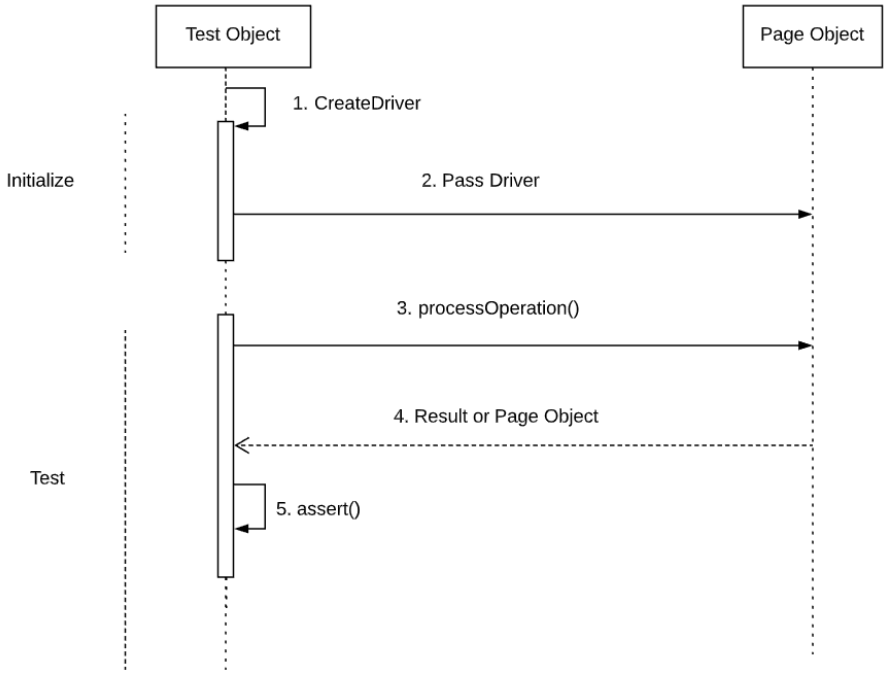

Figure 7. *Interaction between Test and a PageObjec[27]*

According to Figure 7, there is a clear separation between the test object and the page object, such as the locators and the layout. This is how tests and pages are divided in a single repository for the services or operations offered by the page rather than having these services scattered throughout the tests[22]. In the test object, it initializes driver, and tests

are verified by getting UI locators from the page object. This permits changes due to UI changes in both cases to be carried out in one place.

Implementation of POM involves the following steps:

- Review the overall flow of UI Screens.
- Create a Page class for every UI screen.
- A Page Class should return another Page class (via an operation), which represents the next screen in a flow.
- Create Test classes and test methods that perform operations on Page Objects. [27]

To implement that in POM in the above example of a basic search in the Taltech website, we need to create a page class for the homepage, separate locators, and test class for verification. For the Taltech website, we have a login page for login, menu bar for navigation, the main page where user landed, header section for search. Based on these four pages, we can design POM like below:

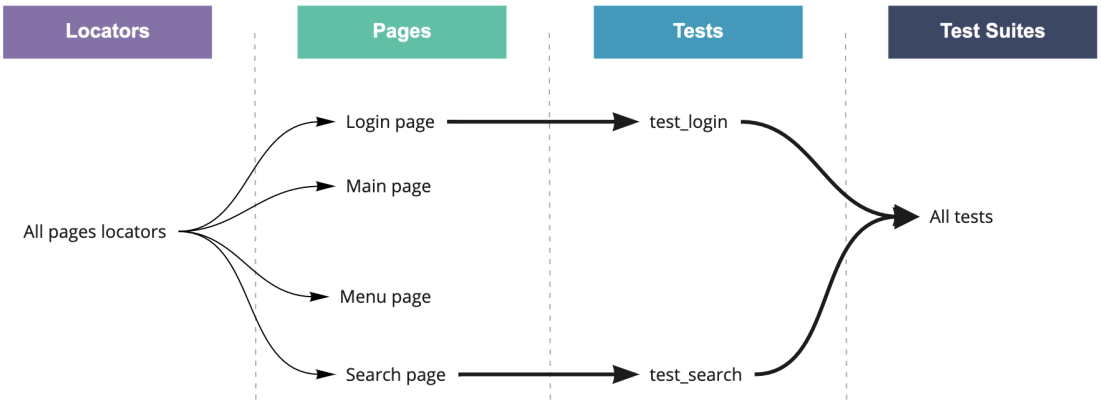

Figure 8. *Page Object Model implementation for Taltech webpage with two tests*

According to Figure 8, Two tests are implemented, but during the test design, we need to consider all the pages. Here, the menu is not a separate page, but I considered it as a page because all the locators of this page need to be separated from the main page, which could be accessed independently in any test.

4.2 Other design patterns

Screenplay model; This model further brings POM into a more legible (and assumed to be maintainable) screenplay structure by arranging page objects, actions, and other elements such as inputs, goals, actors, etc. Screenplay pattern is a template pattern (formerly known as the Journey pattern) for writing acceptance tests that are based on SOLID design principles. SOLID is a part of Object-Oriented Design, specifically:

- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- Interface Substitution Principle
- Dependency Inversion Principle

When writing an acceptance test, For each web page, each page object containing the UI item of the web page and behavior, each entity performs while using the Page Object pattern. The size of the class, therefore, increases with every new item and action. This often leads to an anti-pattern called “Large Class” which is a violation of some SOLID Principles, which are SRP (Single Responsibility Principle) and OCP (Open Closed Principle). SRP states that a class should be responsible for only one responsibility[28]. The OCP notes that an extension of the class should be allowed, but modification should not be allowed. Similar principles are adhered to in the screenplay model, and a separate class is required to perform each actor. This implies that there is a process class for each process. This makes it easier to read and manage a lot of smaller classes instead of a few larger classes.

Façade Design Pattern; It is similar to the Page Object Model, but it’s geared to full facades or shapes where many inputs and possibly more than one action need to happen, the main drawback is that variance in the workflow forces you to create another façade class. The test level calls the entire façade class and provides an object that contains all the inputs needed. It is similar to how API testing is sometimes arranged[28].

Fluent Design Pattern; It is a different flavor of POM that is supposed to conform better with Behavior Driven Development since it forces the test to be done in a "logical chain" or workflow. The page objects are written in a fluent interface manner in which methods can be cascaded or chained in a flow of calls. This is achieved by making the methods return a page class object of the type required to continue the flow [28].

4.3 Dependencies

4.3.1 Python

Python – a programming language that focuses on the readability of code. This is a programming language that lets you work quickly and integrate systems more effectively. Simple to learn, considered to be the best beginner programming language [4]. A strong package management system called *pip* is available, which can be used to install new packages easily on any OSes where Python is available. *pip* installs and updates packages from remote repository called PyPI(Python Package Index) [4]. All packages which are used in this experiment can be acquired via *pip*. Python version 3.8.2 will be used in the experiment of this study.

4.3.2 Selenium Webdriver

As described in chapter 4, Selenium has three components where Selenium Webdriver is used for a robust solution for cross-browser test automation. Selenium Webdriver has bindings for Python, so test cases can be written in a pure programming language using different kinds of helper tools.

Selenium Webdriver controls browser by communicating directly with it; A collection of open-source APIs are used to automation of testing of a web application to verify it works as expected. The interface of Webdriver is the starting point of all Selenium Webdriver API use, where the initial step is to install a WebDriver framework. You create an instance of a WebDriver interface using a browser-specific constructor. The name of instances varies for different browsers and programming languages. For example, invoking a new Chrome instance is like driver = webdriver.Chrome(). The main classes of Selenium Webdriver are: Webdriver and Web element. Webdriver object represents an instance of a real browser's driver and controls the browser's behavior. Web element object represents an element on the web page [4]. Selenium 3.141.0 will be used in this experiment.

4.3.3 Webdriver manager

As we create instances of each browser, the browser driver needs to be in the machine to open it. If the browser driver is not present, Webdriver can't open it. Another way is to provide an executable path for browser driver. For example; driver = webdriver.Chrome("../drivers/chromedriver") where executable path is provided in the parameter. This way is not good solution when you want to run the test in different versions of a

browser. The main idea of WebDriver manger is to simplify the management of binary drivers for different browsers. Currently it supports [29]:

- ChromeDriver
- GeckoDriver
- IEDriver
- OperaDriver
- EdgeChromiumDriver

4.3.4 Unittest

The python version of the unit testing framework; originally inspired by JUnit with a similar flavor of unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework [30]. To achieve this, *unittest* supports some important concepts in an object-oriented way:

test fixture: A test fixture represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process[30].

test case: A test case is the individual unit of testing. It checks for a specific response to a particular set of inputs. *unittest* provides a base class, *TestCase*, which may be used to create new test cases[30].

test suite: A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together[30].

test runner: A test runner is a component that orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests [30].

4.3.5 HTML Reports

Once you have Selenium test suite, it needs to executed and test results needs to be analyzed. On the other hand, stakeholders may want to see reports in a presentable way rather than in console or IDE. As a result, all your test reports need to be organized clearly, so that you have the visibility all at one place. Selenium does not come up with reporting capability, which is one of the most common drawbacks of it. However, there are several third-party tools that can be integrated within Selenium code. For Python, three popular tools are:

- HTMLTestRunner
- Allure
- Nose

I don't need fancy reporting since I won't analyze test run results so that I will use the basic reporting tool HTMLTestRunner.

5. Implementation and Results

For the experiment, I will be using an e-commerce website developed in Shopify. Shopify is a commerce platform that allows anyone to set up an online store and sell their products. Merchants can also sell their products in person with Shopify POS[31]. The core product of Shopify built using Ruby on Rails with data flowing to MySQL. Shopify can be used to sell physical products, digital products, services and consultations, memberships and classes and lessons, etc. The AUT for this experiment is a Tallinn based local shop where the owner wanted to reach customers through online without hampering in-store POS system. As a result, our company has chosen Shopify, where both of the parts can be maintained to meet client expectations. The store sells all kinds of halal foods, including Asian groceries. My experiments will be on the staging website provided by Shopify. This is how the homepage of the website looks like:

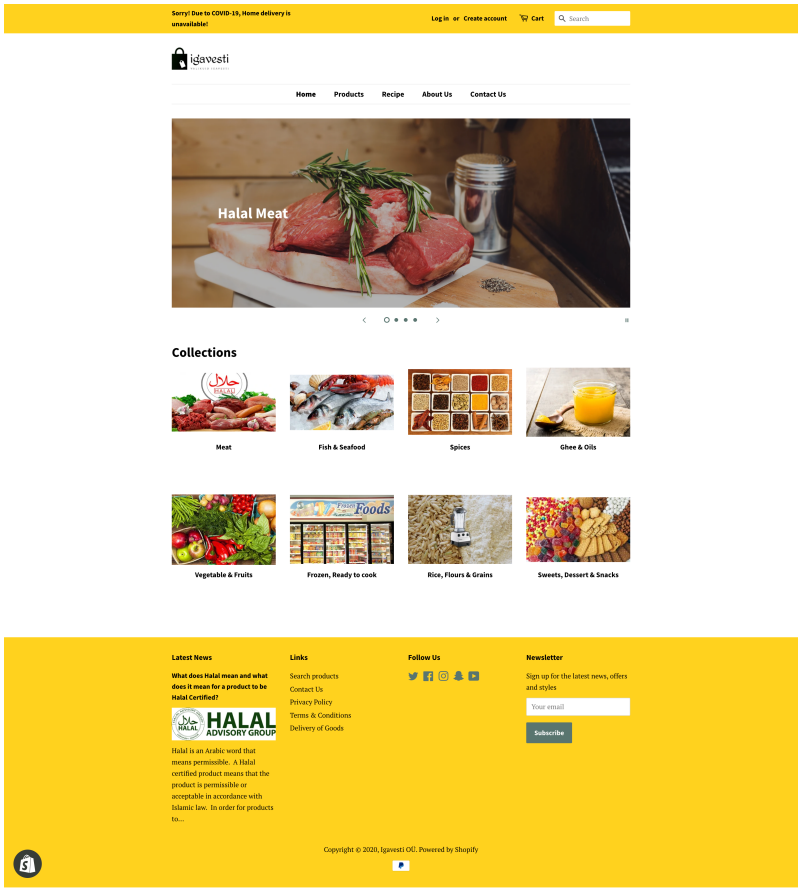

Figure 9. *Homepage of AUT - Igavesti web shop*

According to Figure 9, the mind-map of the website can be implemented from which manual test plans can be derived. The mind map is an easy way to brainstorm thoughts; this is a diagram to display tasks, terms, concepts, or articles related to a central concept or topic, using a non-linear design that enables the user to construct an intuitive context around a fundamental idea. Before moving to test automation implementation, I will illustrate mind-map and document test cases for manual test execution of that website.

Figure 10. *Mind-map of Igavesti*

As illustrated in Figure 10, all the menu and links of the homepage touched at least once, but it can be divided into sibling nodes as well. For example, checkout can be added to the sibling of the cart. As discussed in 2, all test cases can not be implemented for test automation, so from the manual test cases, I will select possible test cases for test automation. Also, my purpose is not to make a robust test automation framework, but I want to experiment maintainability of the test suites when a locator id changes or the website language changes. For test case documentation, items mentioned in the mind-map needs to be covered at least once to fulfill a regression run. Test cases are documented below:

Table 4. *Manual test plan of Igavesti webshop*

Test case ID	**Test scenario**	**Test steps**	**Verification**	**Result**
TI001	Verify header text	1. Go to https://igavesti-ou.myshopify.com 2. Check the header text in top left corner	Header text is shown	Pass
TI002	Check login with valid data	1. Go to https://igavesti-ou.myshopify.com 2. Click on Login 3. Enter valid registered email 4. Enter password 5. Click on Sign In	User is signed in and landed on my account page	Pass
TI003	Check sign up with valid data	1. Go to https://igavesti-ou.myshopify.com 2. Click on Create account 3. Enter first name, last name 4. Enter a valid email and password 5. Click on Create	Account is created successfully.	Pass
TI004	Check empty cart	1. Go to https://igavesti-ou.myshopify.com 2. Click on cart	Your cart is currently empty shown.	Pass
TI005	Add items to the cart	1. Go to https://igavesti-ou.myshopify.com 2. Click on Products 3. Open any product 4. Click on Add to cart	Item is added to the cart, user is taken to Your Cart page. Quantity shows 1	Pass
TI006	Check valid search	1. Go to https://igavesti-ou.myshopify.com 2. Enter a valid search query 3. Press enter from keyboard	Search result shows, heading shows: Your search for "query" revealed the following:	Pass

Table 4 *Test plan continues*

Test case ID	Test scenario	Test steps	Verification	Result
TI007	Verify homepage redirection	1. Go to https://igavesti-ou.myshopify.com 2. Click on cart 3. Click on Home	User is redirected back to home. Collections is shown.	Pass
TI008	Verify products page	1. Stay on any page 2. Click on Products	Products page opened. Section header is: Products, Browse By is shown.	Pass
TI009	Verify recipe page	1. Stay on any page 2. Click on Recipe	Recipe page opened. Section header is : Recipe	Pass
TI010	Verify about us page	1. Stay on any page 2. Click on About Us	About Us page opned. Section header is: About Us	Pass
TI011	Verify contact us submission	1. Stay on any page 2. Click on Contact Us 3. Enter name, email, message 4. Click on send	Success message is shown: Thanks for contacting us. We'll get back to you as soon as possible.	Pass
TI012	Verify all the collections	1. Go to https://igavesti-ou.myshopify.com 2. Click on all the collections	Verify all the collections pages are opening	Pass
TI013	Verify Latest news	1. Go to https://igavesti-ou.myshopify.com 2. Click on Latest news form the footer	News page opened	Pass

Table 4 *Test plan continues*

Test case ID	Test scenario	Test steps	Verification	Result
TI014	Verify links	1. Go to https://igavesti-ou.myshopify.com 2. Click on all the links from the footer: Search products Contact Us Privacy Policy Terms & Conditions Delivery of Goods	All the links pages are opened	Pass
TI015	Verify follow us links	1. Go to https://igavesti-ou.myshopify.com 2. Check Follow Us shows in the footer	Social links are present	Pass
TI016	Verify newsletter sign up	1. Go to https://igavesti-ou.myshopify.com 2. Enter a valid email in newsletter 3. Click on subscribe	Subscribed successfully	Pass
TI017	Verify checkout flow (COD)	1. Go to https://igavesti-ou.myshopify.com 2. Click on Products 3. Open any product 4. Click on Add to cart 5. Click on Checkout 6. Enter email, last name, address, city, postal code 7. Click on Continue to shopping 8. Click on continue to payment 9. Choose Cash on Delivery 10. Complete order	Order is completed, Order ID found.	Pass

Table 4 *Test plan continues*

Test case ID	Test scenario	Test steps	Verification	Result
TI018	Verify checkout flow (Discount code)	1. Go to https://igavesti-ou.myshopify.com 2. Click on Products 3. Open any product 4. Click on Add to cart 5. Click on Checkout 6. Enter email, last name, address, city, postal code 7. Enter discount code 8. Click on Apply 9. Contniue with shopping 10. Continue to payment 11. Complete order	In step 10, it shows Your order is free. No payment is required. Order completed.	Pass

5.1 Proceed to Test Automation

According to the comparison in Table 3, I have decided to use Selenium to implement test automation based on the manual test plan documented in Table 4. This is because selenium is an open-source tool that is cost-effective. In addition to that, it supports almost all popular OS and browsers. Although all the tools compared in Table3 supports record and playback, but I am not implementing record-playback script since it's mostly for non-programmers. As I will be comparing test suites maintainability in Python, Selenium is one of the best tools in terms of available resources with low training costs. If a framework can be implemented in a maintainable way, it would be easier for the client to manage it in the long term. Before moving to write code for test automation, I am following test automation approaches mentioned in Chapter 3, where test case documentation is done in 4. After manual execution, I found in two cases would require a special mechanism to implement using Selenium Webdriver, so these will be skipped in test automation:

- Robot verification (i.e.: reCAPTCHA)
- Paypal checkout

This is because reCAPTCHA is implemented to prevent automated action, so if you can bypass it easily, then it does not make sense to apply it. For Paypal checkout, this is 3rd party web page, and what if the locator changes or the UI changes by them in the future. Things that are not in control within our system, automation won't be reliable there.

For the test suite comparison, I will be using the same parameters in the script developed using POM or without POM. This is about locating elements in a page or type of loop usage in any test method. For the perfect comparison, I will be using the same strategy in both cases.

Selenium automates the browser; using it, you can automate almost every task in the browser as if a real person were to execute the same task. Selenium Webdriver is used to send commands to the browser. As soon as you import Webdriver in your code, you get access to Webdriver API and you can access classes like:

```
webdriver.Chrome
webdriver.Firefox
webdriver.Ie
```

Selenium provides the following methods to locate elements in a page[22]:

- find_element_by_id
- find_element_by_name
- find_element_by_xpath
- find_element_by_link_text
- find_element_by_partial_link_text
- find_element_by_tag_name
- find_element_by_class_name
- find_element_by_css_selector

To find multiple elements (these methods will return a list):

- find_elements_by_name
- find_elements_by_xpath
- find_elements_by_link_text
- find_elements_by_partial_link_text
- find_elements_by_tag_name
- find_elements_by_class_name
- find_elements_by_css_selector

Based on element in the DOM of a webpage, I will be using appropriate ways to locate an element in that webpage. However, the same element can be located in multiple ways. For example: consider this log in and create account page source from the AUT:

```
#Login_email field
  <input type="email" name="customer[email]" id="CustomerEmail"
     placeholder="Email" autocorrect="off" autocapitalize="off"
     autofocus="">
```

```
#Create Account_email field
<input type="email" name="customer[email]" id="Email"
   placeholder="Email" autocorrect="off" autocapitalize="off">
```

For two different element in two different pages; the customer email field has both id and name attribute, so choosing the way of locating this element could be tricky. The id attribute defines a unique ID for an HTML element. The name attribute specifies a name for the element. XPath can be used to navigate through elements and attributes in an XML document [32]. The same element can be located by id, by name, and by XPath, but the preferred way would be by id since its unique. From the above pages, the name is same, but id is different through the field serves the same functionality from two pages. This is how locators can be used for this customer email element:

```
driver.find_element_by_id("CustomerEmail")
driver.find_element_by_name("customer[email]")
driver.find_element_by_xpath("/html/body/main/div/div/div/div/form/input[5]")
```

To automate login scenarios analysing login page HTML attribute would be the first step and choosing the locators.

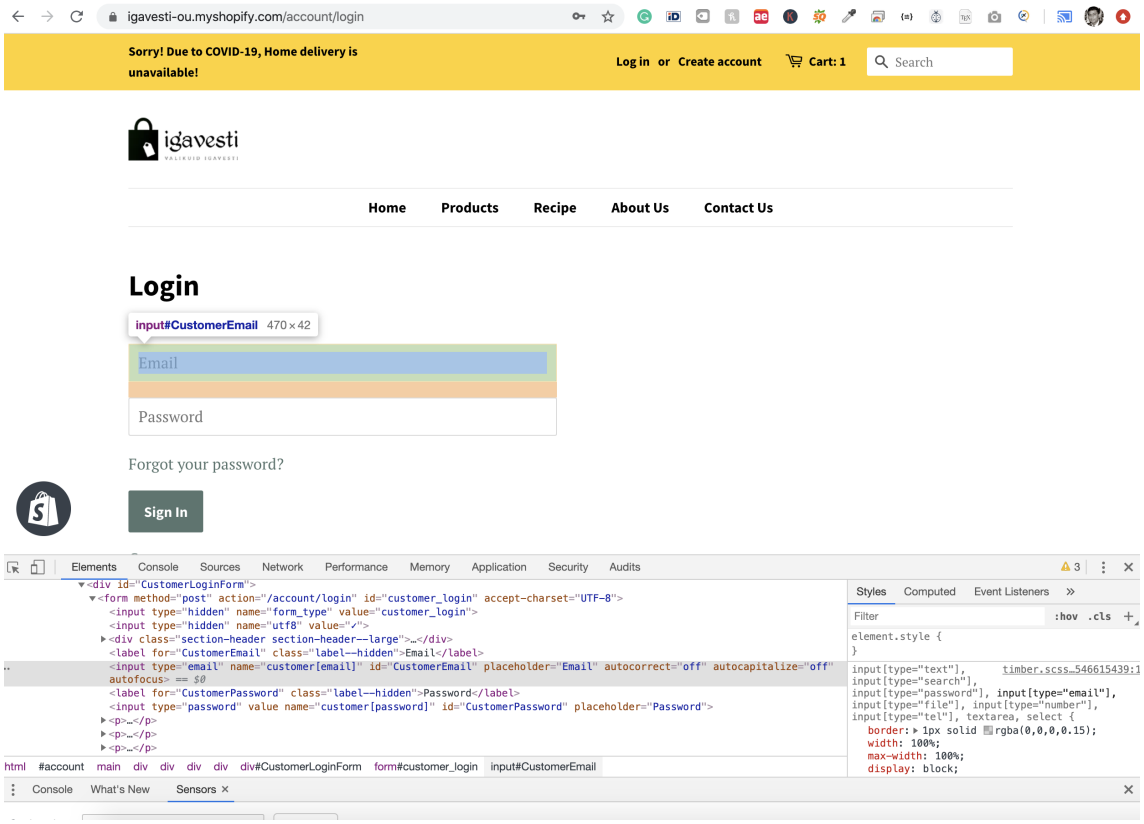

Figure 11. *Igavesti login page UI with HTML elements*

According to Figure 11 Clicking through the desired element and entering inputs would be sufficient to create a single test case. Lets consider checking valid login test from Table 4. The very first task would be to import Webdriver, then open the browser and open the specified URL, click on the login link, enter email, enter the password, and click on the submit button. Finally, quit the browser.

```
from selenium import webdriver # Import webdriver from Selenium
driver = webdriver.Chrome() # Invoke chrome driver
driver.get("https://igavesti-ou.myshopify.com/") # Open the URL
driver.find_element_by_id('customer_login_link').click() # Click on Login link
driver.find_element_by_id("CustomerEmail").send_keys('azadtestlio@gmail.com') # Enter
    the email
driver.find_element_by_id("CustomerPassword").send_keys('Tester1234') # Enter the
    password
driver.find_element_by_xpath("//form[@id='customer_login']//input[@class='btn']").click()
    # Click on submit button
driver.find_element_by_xpath("//h1[contains(text(),'My Account')]").is_displayed() #
    Verify header of the page is My Account
driver.quit() # Quit the browser
```

Now, we have a single test case, but to make it test suite, we need to add more test cases to it. In the above code, opening the URL would be a common case for each test case. Also, quitting the browser needs to happen after executing all test cases. If we start adding these lines in each test cases, the code will become longer and somewhat unmanageable. To organize those tests accurately and run them all together, unitest framework comes as a solution. Python unittest module is used to test a unit of source code, but this borrows four basic concepts from the unit testing which are: test fixture, test case, test suite, test runner mentioned in 4.3.4. Unittest is built into the Python; it contains both a testing framework and a test runner. Unittest requires to put tests into classes as methods and use assertion methods in the unittest.TestCase class instead of the built-in assert statement. The structure of unitest with a couple of tests looks like below:

```
import unittest

class TestSuite(unittest.TestCase):
    def setUp(self):
        ...
    @classmethod
    def setUpClass(self):
        ...
    def test_case_01(self):
        ...
    def test_case_02(self):
        ...
    def tearDown(self):
        ...
    @classmethod
    def tearDownClass(self):
        ...
```

setUp() : Method called to prepare the test fixture. This is called immediately before calling the test method; other than AssertionError or SkipTest, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing[30].

tearDown() : Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state [30].

setUpClass() : A class method called before tests in an individual class are run. setUpClass is called with the class as the only argument and must be decorated as a classmethod() [30].

tearDownClass() : A class method called after tests in an individual class have run. tearDownClass is called with the class as the only argument and must be decorated as a classmethod() [30]. A classmethod transform a method into a class method.

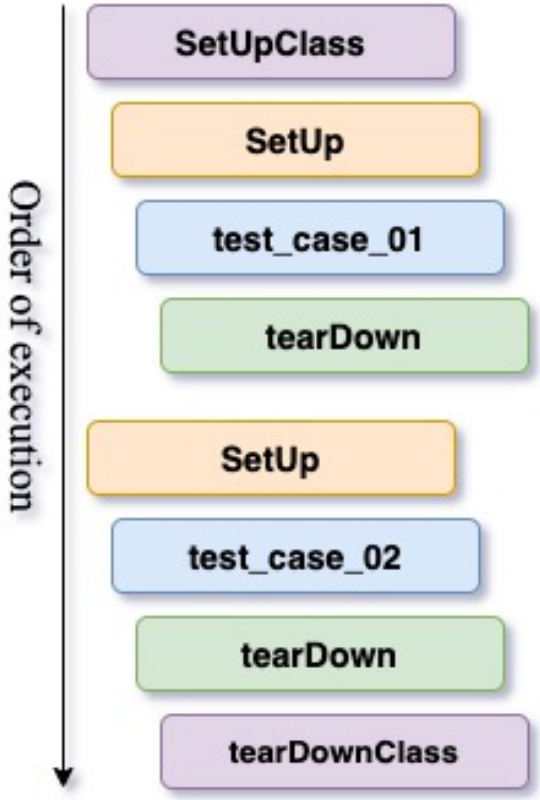

Figure 12. *Order of execution for the example test cases*

Figure 12 is the illustration of the order of the execution for the test suite in the unittest with an example of a couple of tests. Now, converting actual test cases to a unittest test case, the following steps need to be accomplished:

1. Import unittest
2. Create a class called TestLogin
3. Add test into methods of the class
4. Add the command-line entry point to call unittest.main()

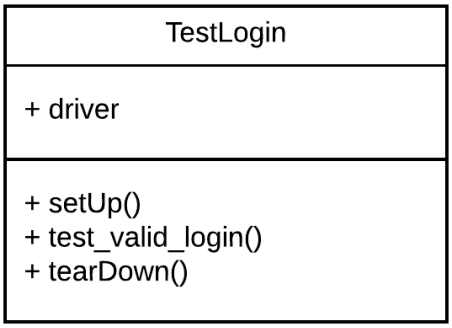

Figure 13. *A single login test case class diagram*

In the Figure 13, I used setUp and tearDown methods as there is only one test case. When designing a test suite, some actions needs to be done initially and use it throughout the test execution. For example, test URL opening requires once before start execution, so it can be added to setUpClass. The same applies for quit a browser once all the test finished, so tearDownClass is the perfect choice when adding more tests. Multiple test cases can be added in the same suite by adding more methods:

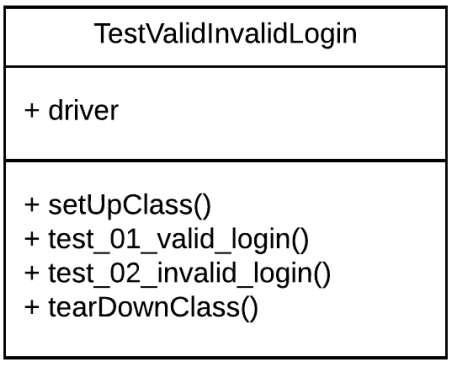

Figure 14. *Class diagram of login test cases without POM*

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to setUpClass being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run[30]. Tests are alphabetically ordered, so I added 01 and 02 for ordering these tests showed in Figure 14. However, there are several ways to order tests, add @ordered tag before each test case. By adding all the test cases mentioned in Igavesti test plan in Table 4, this becomes the test suite without Page Object Model. The full code can be viewed in appendix in the chapter 6

5.2 Test suite with Page Object Model

The same test suite implemented using Page Object Model where tests are separated and pages are separated. All the locators are added in a separate file as well. As a result, if any locator id changes, it can be modified from one place. The above tests for login can be implemented in Page Object Model. In POM, every web page should have a separate class having its objects and methods, then test script should be separated from the object. Here, we have three pages related to the tests: My account page for assertion after successful login, Homepage where login link clicked and Login page where email, password submitted, . No need touch setUpClass and tearDownClass since these does not contain any page information.

In the homepage, finding login link and clicking on it are the tasks so here one object and one action needs to be created. In the login page class, three objects and three methods needs to be created where locators of email field,password field, sign in button are objects and methods are entering email, password, clicking on sign in button. Finally, in the my account page class, two locators and two method needs to be created where getting the header text and click on logout link are methods and locators of those elements are the objects. All the pages are separated from the tests, now test needs to be modified to use attributes of these classes.

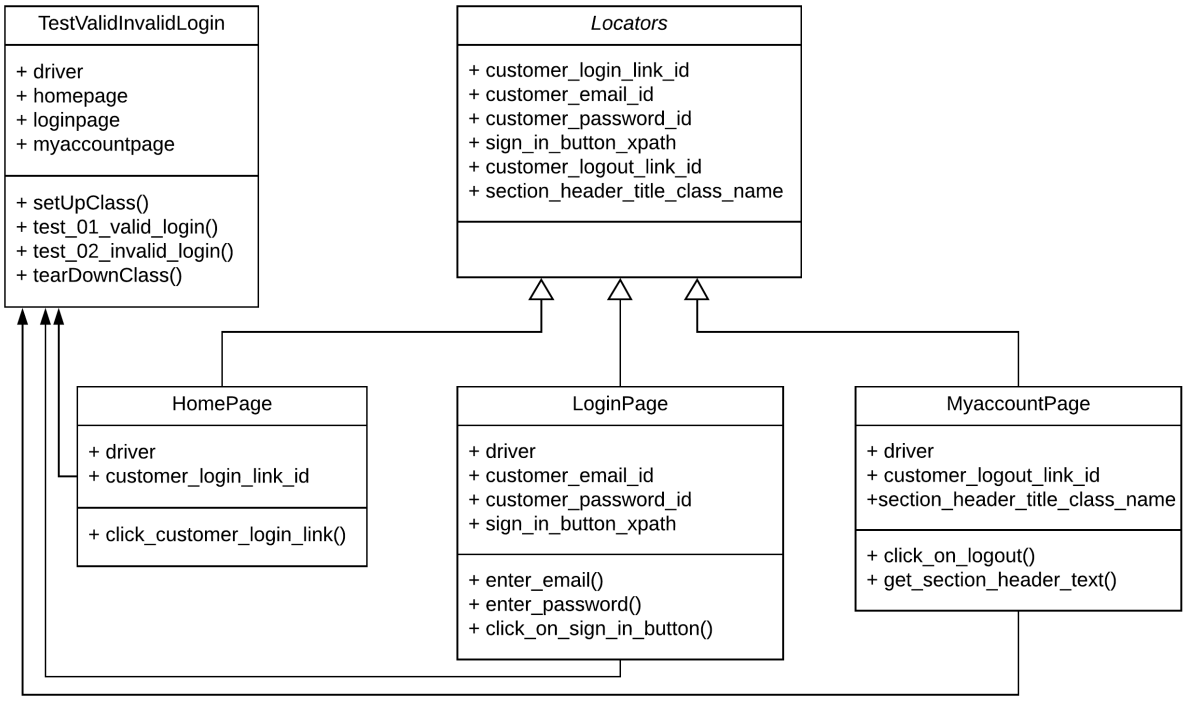

Figure 15. *Class diagram of valid invalid login test cases with POM*

This class diagram are for two test cases but for my analysis I have created a full test suite with all 18 test cases mentioned in Table 4. The full code can be viewed in the appendix in chapter 6 . The source code tree of the codes without POM and with POM are generated below:

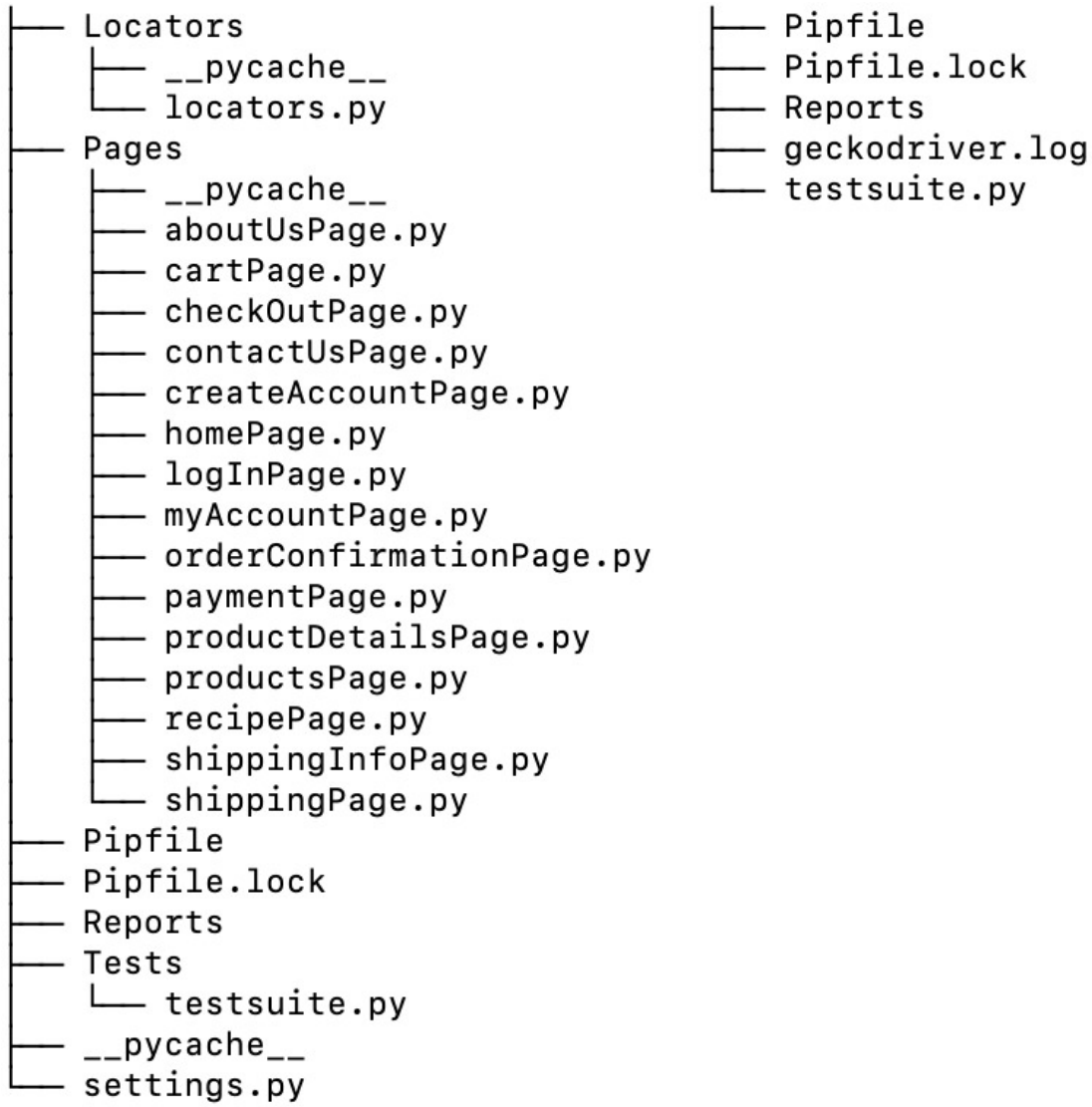

Figure 16. *Source code tree of test suites, With POM on the left and Without POM on the right*

According to Figure 16 with POM code tree has many folders and files but only a few when doing the same without POM. However, all the locators are within test cases in the code without POM which makes difficult to maintain it.

5.3 Analysis

A total 44 locators used in the test suites in both of the codes for the same test plan. Analysis of locators from each pages are listed below:

Table 5. *Homepage locators*

Homepage locators	**Used in Places**		**Places need to be changed**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
header_text_xpath	1	1	1	1
customer_login_link_id	2	2	2	1
customer_register_link_id	1	1	1	1
cart_class_name	1	1	1	1
search_field_xpath	1	1	1	1
latest_news_link_text	1	1	1	1
links_partial_link_text	2	2	2	1
social_links_xpath	1	1	1	1
subscribe_email_field_id	1	1	1	1
subscribe_button_id	1	1	1	1
subscribe_success_css_selector	1	1	1	1
home_link_text	17	17	17	1
products_link_text	5	5	5	1
recipe_link_text	1	1	1	1
about_us_link_text	1	1	1	1
contact_us_xpath	1	1	1	1

Table 6. *My account page locators*

My Account pages locators	**Used in Places**		**Places need to be changed**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
my_account_header_xpath	1	1	1	1
account_details_xpath	1	1	1	1
customer_log_out_link_id	1	1	1	1

Table 7. *Login & Sign Up page locators*

Login & Sign up page locators	**Used in Places**		**Places need to be changed**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
customer_email_id	3	3	3	1
customer_password_id	2	2	2	1
sign_in_button_xpath	2	2	2	1
first_name_id	1	1	1	1
last_name_id	1	1	1	1
email_id	3	3	3	1
create_password_id	1	1	1	1
create_button_xpath	1	1	1	1

Table 8. *Product flow pages locators*

Product & PDP pages locators	**Used in Places**		**Places need to be changed**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
all_items_css_selector	3	3	3	1
add_to_cart_button_id	3	3	3	1

Table 9. *Cart page locators*

Cart page locators	**Used in Places**		**Places need to be changed**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
remove_cart_class_name	1	1	1	1
checkout_button_name	2	2	2	1

Table 10. *Contact us page locators*

Contact us page locators	**Used in Places**		**Places need to be changed**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
contact_form_name_id	1	1	1	1
contact_form_email_id	1	1	1	1
contact_form_message_id	1	1	1	1
contact_send_button_xpath	1	1	1	1

Table 11. *Shipping flow pages locators*

Shipping flow pages locators	**Used in Places**		**Need changes**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
checkout_email_or_phone_xpath	2	2	2	1
checkout_shipping_address_last_name_id	2	2	2	1
checkout_shipping_address_address1_id	2	2	2	1
checkout_shipping_address_city_id	2	2	2	1
checkout_shipping_address_zip_id	2	2	2	1
shipping_checkout_continue_button_xpath	4	4	4	1
checkout_reduction_code_id	2	2	2	1
continue_to_payment_xpath	4	4	4	1
cash_on_delivery_radio_button_xpath	1	1	1	1
complete_order_button_id	4	4	4	1
order_number_class_name	2	2	2	1
continue_shopping_link_text	1	1	1	1

For the analysis, I am excluding the locators which used only once. Combining all the locators which are used more than once.

Table 12. *All pages locators which were used at least twice*

All pages locators used multiple times	**Used in Places**		**Change in places**	
	POM No	**POM Yes**	**POM No**	**POM Yes**
customer_login_link_id	2	2	2	1
links_partial_link_text	2	2	2	1
home_link_text	17	17	17	1
products_link_text	5	5	5	1
customer_email_id	3	3	3	1
customer_password_id	2	2	2	1
sign_in_button_xpath	2	2	2	1
email_id	3	3	3	1
all_items_css_selector	3	3	3	1
add_to_cart_button_id	3	3	3	1
checkout_button_name	2	2	2	1
checkout_email_or_phone_xpath	2	2	2	1
checkout_shipping_address_last_name_id	2	2	2	1
checkout_shipping_address_address1_id	2	2	2	1
checkout_shipping_address_city_id	2	2	2	1
checkout_shipping_address_zip_id	2	2	2	1
shipping_checkout_continue_button_xpath	4	4	4	1
checkout_reduction_code_id	2	2	2	1
continue_to_payment_xpath	4	4	4	1
complete_order_button_id	4	4	4	1
order_number_class_name	2	2	2	1

From the Table 12 the plotting can be drawn below:

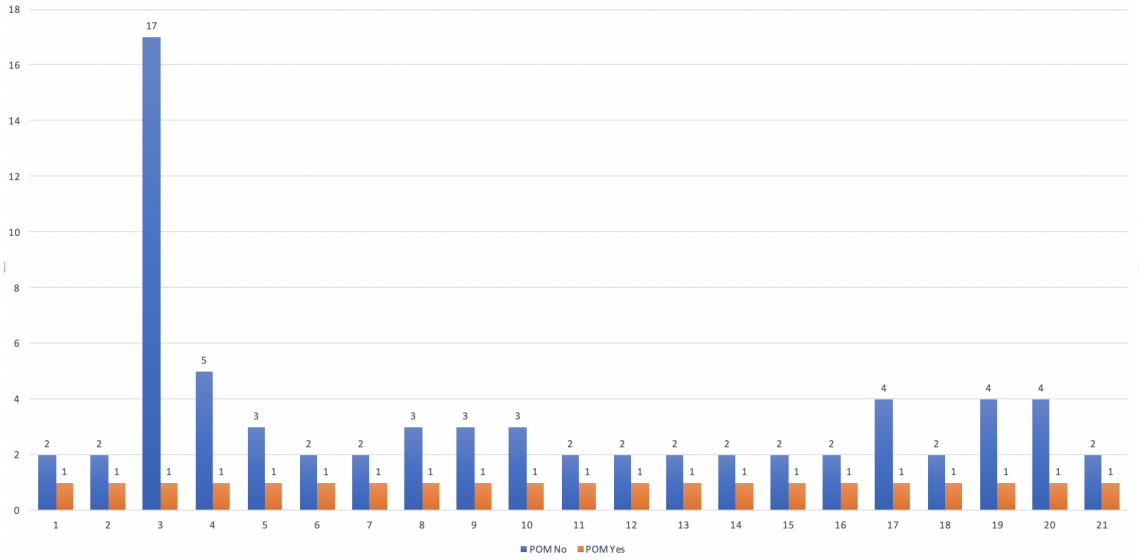

Figure 17. *All pages locators- Changes requires for each id changes*

According to Figure 17, it can be seen that for home link text only 1 changes required in the test suite designed with POM, whereas 17 changes required for the same id. As this is small test suite, not many locators used multiple times but at least 8 times more changes required as per the results of the [3] which can be changed for big test suite with at least 100 test cases where tests are related to each other.

If website language changes, then the locators with id won't need to changes because these are unique. However, link_text, partial_link_text, name, xpath etc. needs to changes. In that case, less locators needs to be changes but the number of places needs remains same.

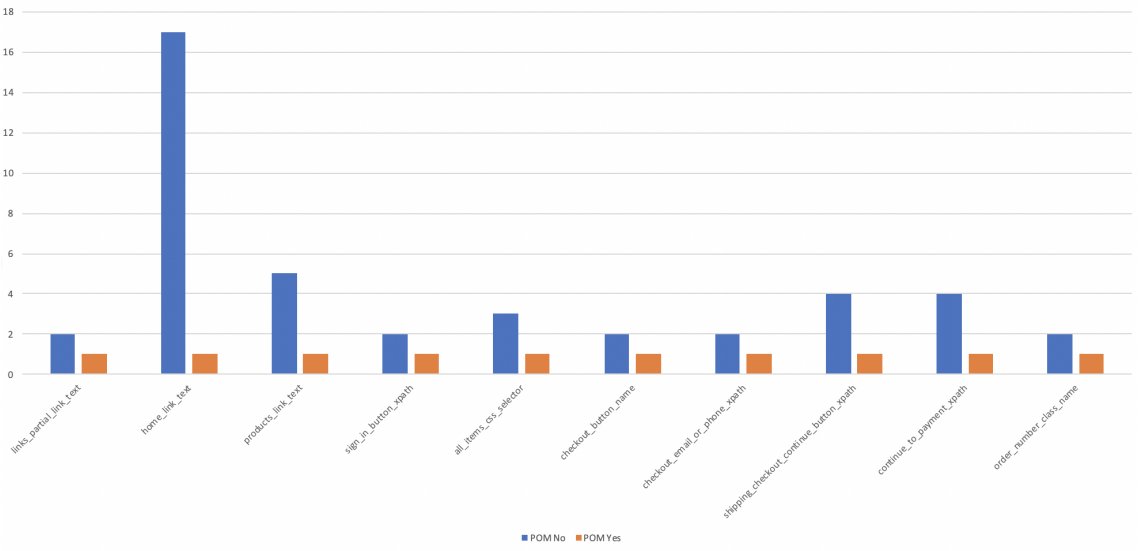

Figure 18. *All pages locators- Changes requires for language changes*

6. Summary

Web test automation is becoming complex due to dynamic web development, but having a test suite by investing time and money can be fruitful in the long run. The results of this experimental study conducted to check how POM can improve maintainability in test automation suites. I can give a positive answer to my research questions where the test suite implemented using POM requires less effort to update when a locator id changes or the webpage language changes. After this experiment, I came to a solution to keep these points in mind when designing test suite:

- Consider test automation as a software development
- Choose the way of locator which has less chance to be changed
- Test data should be in a separate file
- All the locators should be in a single file
- Common tasks in the SetUpClass
- Last tasks in tearDownClass

Due to time constraints, I could not integrate the tests CI/CD, but this can be done in the future. Finally, to get the best result of test automation, it must be designed in a way that maintenance of the test suite takes less time, less effort and adopt with changes by keeping the same automation tool.

After doing this thesis, my experience with programming has increased. Test automation with CI/CD process are in my portfolio now, so that I can look forward to the next challenge in my career from manual to automation testing.

Bibliography

[1] Monika Sharma and Rigzin Angmo. "Web based Automation Testing and Tools". In: *(IJCSIT) International Journal of Computer Science and Information Technologies* 5.1 (2014), pp. 908–912.

[2] Børge Haugset and Geir Kjetil Hanssen. "Automated Acceptance Testing: A Literature Review and an Industrial Case Study". In: *Agile 2008 Conference* (2008), pp. 27–38. DOI: 10.1109/agile.2008.82.

[3] Maurizio Leotta et al. "Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* (2013), pp. 108–113. DOI: 10.1109/icstw.2013.19.

[4] Kaarel Allik. "Selenium-Based web Test Automation Framework Development". 2015.

[5] Maurizio Leotta et al. "Automated generation of visual web tests from DOM-based web tests". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC 15* (2015). DOI: 10.1145/2695664.2695847.

[6] Cem Kaner. *Improving the Maintainability of Automated Test Suites*.

[7] Amorim Daniel. *Why companies choose paid test automation tools rather than free? | LinkedIn*. https://www.linkedin.com/pulse/why-companies-choose-paid-test-automation-tools-rather-daniel-amorim/. (Accessed on 04/10/2020). Apr. 2018.

[8] Joost Visser. *Building maintainable software: ten guidelines for future-proof code*. OReilly, 2016.

[9] Muhammad Abid Jamil et al. "Software Testing Techniques: A Literature Review". In: *2016 6th International Conference on Information and Communication Technology for The Muslim World* 16.1 (2016), pp. 177–182.

[10] Juraj Húska. "Automated Testing of the Component-based Web Application User Interfaces". 2012.

[11] "ISO/IEC/IEEE International Standard - Software and systems engineering – Software testing –Part 3: Test documentation". In: *ISO/IEC/IEEE 29119-3:2013(E)* (2013), pp. 1–138.

[12] *Continuous integration - Wikipedia*. https://en.wikipedia.org/wiki/Continuous_integration. (Accessed on 04/17/2020).

[13] *Best Practices for Continuous Deployment | Lucidchart Blog*. https://www.lucidchart.com/blog/continuous-deployment-best-practices. (Accessed on 04/20/2020).

[14] *Continuous Integration*. https://martinfowler.com/articles/continuousIntegration.html. (Accessed on 04/21/2020).

[15] Vahid Garousi and Frank Elberzhager. "Test Automation: Not Just for Test Execution". In: *IEEE Software* 34.2 (2017), pp. 90–96. DOI: 10.1109/ms.2017.34.

[16] Heidilyn Veloso Gamido and Marlon Viray Gamido. "Comparative Review of the Features of Automated Software Testing Tools". In: *International Journal of Electrical and Computer Engineering (IJECE)* 9.5 (Jan. 2019), pp. 4473–4478. DOI: 10.11591/ijece.v9i5.pp4473-4478.

[17] Harish Mittal. "Comparative Analysis of Automated Functional Testing Tools". In: *Journal of Network Communications and Emerging Technologies (JNCET)* 6.6 (June 2016), pp. 50–53. DOI: 10.26565/2519-2310-2019-1-07.

[18] Majid Khan et al. "Comparative Analysis of Automated Software Testing Tools". In: *International Journal of Soft Computing and Engineering (IJSCE)* 6.4 (Sept. 2016), pp. 46–49. DOI: 10.26565/2519-2310-2019-1-07.

[19] F. Okezie, I. Odun-Ayo, and S. Bogle. "A Critical Analysis of Software Testing Tools". In: *Journal of Physics: Conference Series* 1378 (Dec. 2019), p. 042030. DOI: 10.1088/1742-6596/1378/4/042030. URL: https://doi.org/10.1088%2F1742-6596%2F1378%2F4%2F042030.

[20] Inderjeet Singh and Bindia Tarika. "Comparative Analysis of Open Source Automated Software Testing Tools: Selenium, Sikuli and Watir". In: *International Journal of Information Computation Technology* 4.15 (2014), pp. 1507–1518.

[21] *Selenium (software) - Wikipedia*. https://en.wikipedia.org/wiki/Selenium_(software). (Accessed on 04/27/2020).

[22] *SeleniumHQ Browser Automation*. https://www.selenium.dev/. (Accessed on 04/27/2020).

[23] *Watir - Wikipedia*. https://en.wikipedia.org/wiki/Watir. (Accessed on 04/27/2020).

[24] *TestComplete - Wikipedia*. https://en.wikipedia.org/wiki/TestComplete. (Accessed on 04/27/2020).

[25] *Ranorex Studio - Wikipedia*. https://en.wikipedia.org/wiki/Ranorex_Studio. (Accessed on 04/27/2020).

[26] *LoadRunner - Wikipedia*. https://en.wikipedia.org/wiki/LoadRunner. (Accessed on 04/27/2020).

[27] *Design Patterns - Page Object Model - Experitest - Test Execution*. https://docs.experitest.com/display/TE/Design+Patterns+-+Page+Object+Model. (Accessed on 04/28/2020).

[28] *UI Automation - Page Object Model and other Design Patterns - Microsoft Tech Community - 992242*. https://techcommunity.microsoft.com/t5/testingspot-blog/ui-automation-page-object-model-and-other-design-patterns/ba-p/992242?fbclid=IwAR2KGUiJ_MKsjJuk_2tGTEAQvkGaNsJzw9xxEDY91uaYlefkpRNbG2jI1GY#. (Accessed on 05/04/2020).

[29] *SergeyPirogov/webdriver_manager*. https://github.com/SergeyPirogov/webdriver_manager. (Accessed on 05/06/2020).

[30] *unittest — Unit testing framework — Python 3.8.3rc1 documentation*. https://docs.python.org/3/library/unittest.html. (Accessed on 05/06/2020).

[31] *What is Shopify? How to Start Selling on Shopify*. https://www.shopify.com/blog/what-is-shopify. (Accessed on 05/05/2020).

[32] *HTML name Attribute*. https://www.w3schools.com/tags/att_name.asp. (Accessed on 05/11/2020).

Appendices

Appendix 1 - Full test suite without POM

Github link: `https://github.com/azadnsu/IgavestiWithoutPOM`

Valid login test:

```
from selenium import webdriver # Import webdriver from Selenium
import unittest # Import unittest

class TestLogin( unittest .TestCase):

    def setUp( self ):
        self . driver = webdriver.Chrome() # Invoke chrome driver

    def test_valid_login ( self ):
        driver = self . driver # Setup driver variable to self . driver so no need to
            type self . driver each time
        driver . get(" https :// igavesti -ou.myshopify.com/") # Open the URL
        driver . find_element_by_id(' customer_login_link ' ). click () # Click on Login
            link
        driver . find_element_by_id("CustomerEmail").
        send_keys(' azadtestlio@gmail . com') # Enter the email
        driver . find_element_by_id("CustomerPassword").
        send_keys('Tester1234') # Enter the password
        driver . find_element_by_xpath(
            " // form[@id='customer_login ']// input [@class='btn']"). click () # Click on
                submit button
        section_header =
        driver . find_element_by_class_name(" section - header__title ") # Find the section
            header
        self . assertEqual ( section_header . text , "My Account") # Verify header of the
            page is My Account
```

```
    def tearDown(self):
        self.driver.quit() # Quit the browser

if __name__ == '__main__':
    unittest.main()
```

Valid invalid login tests:

```
from selenium import webdriver # Import webdriver from Selenium
import unittest # Import unittest

class TestLogin(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        cls.driver = webdriver.Firefox() # Invoke chrome driver just at the begining
            of test execution
        cls.driver.get("https://igavesti-ou.myshopify.com/") # Open the URL at the
            beginning of test

    def test_01_valid_login(self):
        driver = self.driver # Setup driver variable to self.driver so no need to
            type self.driver each time
        driver.find_element_by_id('customer_login_link').click() # Click on Login
            link
        driver.find_element_by_id("CustomerEmail").
        send_keys('azadtestlio@gmail.com') # Enter the email
        driver.find_element_by_id("CustomerPassword").
        send_keys('Tester1234') # Enter the password
        driver.find_element_by_xpath(
            "//form[@id='customer_login']//input[@class='btn']").click() # Click on
                submit button
        section_header =
        driver.find_element_by_class_name("section-header__title") # Find the section
            header
        self.assertEqual(section_header.text, "My Account") # Verify header of the
            page is My Account
        driver.find_element_by_id('customer_logout_link').click() # Need to sign out
            after this test for the next one
```

```
    def test_02_invalid_login (self):
        driver = self.driver # Setup driver variable to self.driver so no need to
            type self.driver each time
        driver.find_element_by_id('customer_login_link').click() # Click on Login
            link
        driver.find_element_by_id("CustomerEmail").
        send_keys('azadtestlio@gmail.com') # Enter the email
        driver.find_element_by_id("CustomerPassword").
        send_keys('Tester12345') # Enter the password
        driver.find_element_by_xpath(
            "//form[@id='customer_login']//input[@class='btn']").click() # Click on
                submit button
        assert 'Incorrect email or password.' in driver.page_source # Verify error is
            shown

    @classmethod
    def tearDownClass(cls):
        cls.driver.quit() # Quit the browser

if __name__ == '__main__':
    unittest.main()
```

```
import unittest
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
import time
import HtmlTestRunner
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import random
from selenium.webdriver.common.keys import Keys
from selenium.common.exceptions import NoSuchElementException
from random import randint

class IgaveestiTestSuite (unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        cls.driver = webdriver.Chrome(ChromeDriverManager().install())
```

```
        cls . driver .  implicitly_wait  (10)
        cls . driver . maximize_window()

    def   test_01_header_text_present  ( self ):
        driver  =  self . driver
        driver . get (" https  ://  igavesti  -ou.myshopify.com/")
        driver . find_element_by_xpath(" // div[ @class='header-bar__module
            header-bar__message']").is_displayed()
        time . sleep (1)

    @unittest . skip("Due to  reCAPTCHA skip it")
    def   test_02_valid_login  ( self ):
        driver  =self . driver
        Login_link  =  driver . find_element_by_id(' customer_login_link ' )
        Login_link . click ()
        WebDriverWait(driver,  10). until (
            EC.presence_of_element_located ((By.ID,  "CustomerEmail"))
        )
        driver . find_element_by_id("CustomerEmail").send_keys('azadtestlio@gmail .com')
        driver . find_element_by_id("CustomerPassword").send_keys('Tester1234' )
        driver . find_element_by_xpath(" // form[@id='customer_login ']// input [@class='btn']") . click ()
        driver . find_element_by_xpath(" // h1[ contains ( text () ,' My
            Account')]"). is_displayed ()
        driver . find_element_by_xpath(" // h2[ contains ( text () ,' Account
            Details ') ]"). is_displayed ()
        driver . find_element_by_id(' customer_logout_link ' ) . click ()

    @unittest . skip("Due to  reCAPTCHA skip it")
    def   test_002_invalid_login  ( self ):
        driver  =self . driver
        Login_link  =  driver . find_element_by_id(' customer_login_link ' )
        Login_link . click ()
        driver . find_element_by_id("CustomerEmail").send_keys('azadtestlioyy@gmail .com')
        driver . find_element_by_id("CustomerPassword").send_keys('Tester1234' )
        driver . find_element_by_xpath(" // form[@id='customer_login ']// input [@class='btn']") . click ()
        assert  ' Incorrect  email  or  password.'  in  driver . page_source

    @unittest . skip("Due to  reCAPTCHA skip it")
    def   test_03_create_account  ( self ):
        driver  =  self . driver
        driver . find_element_by_id(" customer_register_link  "). click ()
```

```
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "btn"))
        )
        name_generator = 'Azad'+str(random.randint(0, 99))
        password_generator = 'Tester'+str(random.randint(0, 99))
        driver.find_element_by_id("FirstName").send_keys(name_generator)
        driver.find_element_by_id("LastName").send_keys(name_generator)
        driver.find_element_by_id("Email").send_keys(name_generator+'@gmail.com')
        driver.find_element_by_id("CreatePassword").send_keys(password_generator)
        driver.find_element_by_xpath("//form[@id='create_customer']//input[@class='btn']").click()
        time.sleep(2)

    def test_04_empty_cart(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        driver.find_element_by_class_name("cart-page-link").click()
        assert "Your cart is currently empty" in driver.page_source
        assert driver.find_elements_by_css_selector("p.cart--empty-message") ==
            "Your cart is currently empty."

    def test_05_add_product_to_cart(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        driver.find_element_by_link_text('Products').click()
        all_items = driver.find_elements_by_css_selector('p.grid-link__title')
        item = all_items[randint(0, len(all_items)-1)]
        print(item.text)
        item.click()
        driver.find_element_by_id('AddToCart').click()
        assert "Your Shopping Cart" in driver.title
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "btn"))
        )
        driver.find_element_by_class_name('cart__remove').click()
        assert "Your cart is currently empty" in driver.page_source

    def test_06_valid_search(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        search_term = 'Chicken'
        search_field = driver.find_element_by_xpath("//div[@class='header-bar__right
```

```
            post-large--display-table-cell']//input[@placeholder='Search']")
        search_field.send_keys(search_term)
        search_field.send_keys(Keys.RETURN)
        try:
            assert search_term in driver.title
            print("Assertion Test Passed")
        except Exception as e:
            print("Assertion Test Failed", format(e))

    def test_07_homepage_redirection(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        assert 'Home' in driver.title

    def test_08_products_page_redirection(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        driver.find_element_by_link_text('Products').click()
        assert 'Products' in driver.title


    def test_09_recipe_page_redirection(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        driver.find_element_by_link_text('Recipe').click()
        assert 'Recipe' in driver.title

    def test_10_about_us_page_redirection(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        driver.find_element_by_link_text('About Us').click()
        assert 'About Us' in driver.title

    @unittest.skip("Due to reCAPTCHA skip it")
    def test_11_contact_us_form_submission(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        driver.find_element_by_xpath("//a[@class='site-nav__link'][contains(text(),'Contact
            Us')]").click()
        name_generator = 'Azad' + str(random.randint(0, 99))
        driver.find_element_by_id('ContactFormName').send_keys(name_generator)
```

```
        driver.find_element_by_id('ContactFormEmail').send_keys(name_generator+'@gmail.com')
        driver.find_element_by_id('ContactFormMessage').send_keys('Test message,
            please ignore')
        driver.find_element_by_xpath("//input[@class='btn right']").click()
        assert "Thanks for contacting us. We'll get back to you as soon as possible."
            in driver.page_source

    def test_12_all_collections(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        collections = ["Meat", "Fish", "Spices", "Ghee", "Vegetable", "Frozen",
            "Rice", "Sweets"]
        for collection_name in collections:
            driver.find_element_by_link_text('Home').click()
            driver.find_element_by_partial_link_text(collection_name).click()
            assert collection_name in driver.title

    def test_13_latest_news(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        driver.find_element_by_link_text('Latest News').click()
        assert 'News' in driver.title

    def test_14_links_from_footer(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        footer_partial_link_text = ["Search", "Contact", "Privacy", "Terms",
            "Delivery"]
        for link in footer_partial_link_text:
            driver.find_element_by_link_text('Home').click()
            driver.find_element_by_partial_link_text(link).click()
            assert link in driver.title

    def test_15_follow_us_from_footer(self):
        driver = self.driver
        driver.find_element_by_link_text('Home').click()
        try:
            driver.find_elements_by_xpath("//ul[@class='inline-list social-icons']")
        except NoSuchElementException:
            return False
        return True
```

```
@unittest.skip("Due to reCAPTCHA skip it")
def test_16_subscribe(self):
    driver = self.driver
    driver.find_element_by_link_text('Home').click()
    email_generator = 'username'+str(random.randint(0, 999))+'@gmail.com'
    driver.find_element_by_id('Email').clear()
    driver.find_element_by_id('Email').send_keys(email_generator)
    driver.find_element_by_id('subscribe').click()
    if driver.find_elements_by_css_selector('p.note form-success'):
        print("Element exists")
    else:
        print("No such element exist")


def test_17_complete_checkout_cash_on_delivery(self):
    driver = self.driver
    driver.find_element_by_xpath("//a[@class='site-nav__link'
    [contains(text(),'Products')]").click()
    all_items = driver.find_elements_by_css_selector('p.grid-link__title')
    item = all_items[randint(0, len(all_items) - 1)]
    print(item.text)
    item.click()
    driver.find_element_by_id('AddToCartText').click()
    driver.find_element_by_name('checkout').click()
    WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "checkout_email_or_phone"))
    )
    driver.find_element_by_xpath("//input[@id='checkout_email_or_phone']").
    send_keys('username'+str(random.randint(0,999))+'@gmail.com')
    driver.find_element_by_id('checkout_shipping_address_last_name').
    send_keys('LastName'+str(random.randint(0,99)))
    driver.find_element_by_id('checkout_shipping_address_address1').
    send_keys('Tester Lane'+str(random.randint(0,99)))
    city = ["Tallinn", "Tartu", "Parnu"]
    driver.find_element_by_id('checkout_shipping_address_city').send_keys(random.choice(city))
    driver.find_element_by_id('checkout_shipping_address_zip').send_keys(random.randint(10000,15000)
    driver.find_element_by_xpath("//button[@id='continue_button']").click()
    WebDriverWait(driver, 10).until(
        EC.element_to_be_clickable((By.ID, "continue_button"))
    )
```

```
        driver.find_element_by_xpath("//button[@id='continue_button']").click()

        driver.find_element_by_xpath("//input[@id='checkout_payment_gateway_47481028746']").click()
        time.sleep(1)
        driver.find_element_by_id("continue_button").click()
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "os-order-number"))
        )
        assert 'Thank you for your purchase' in driver.title
        print(driver.find_element_by_class_name('os-order-number'))
        driver.find_element_by_link_text("Continue shopping").click()

    def test_18_complete_checkout_by_discount_code(self):
        driver = self.driver
        driver.find_element_by_xpath("//a[@class='site-nav__link'][contains(text(),'Products')]").click(
        all_items = driver.find_elements_by_css_selector('p.grid-link__title')
        item = all_items[randint(0, len(all_items) - 1)]
        print(item.text)
        item.click()
        driver.find_element_by_id('AddToCartText').click()
        driver.find_element_by_name('checkout').click()
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.ID, "checkout_email_or_phone"))
        )
        driver.find_element_by_id('checkout_reduction_code').send_keys('TALTECH')
        driver.find_element_by_id('checkout_reduction_code').send_keys(Keys.RETURN)
        driver.find_element_by_xpath("//input[@id='checkout_email_or_phone']").
        send_keys('username'+str(random.randint(0,999))+'@gmail.com')
        driver.find_element_by_id('checkout_shipping_address_last_name').
        send_keys('LastName'+str(random.randint(0,99)))
        driver.find_element_by_id('checkout_shipping_address_address1').
        send_keys('Tester Lane'+str(random.randint(0,99)))
        city = ["Tallinn", "Tartu", "Parnu"]
        driver.find_element_by_id('checkout_shipping_address_city').
        send_keys(random.choice(city))
        driver.find_element_by_id('checkout_shipping_address_zip').
        send_keys(random.randint(10000,15000))

        driver.find_element_by_xpath("//button[@id='continue_button']").click()
        WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.ID, "continue_button"))
```

```
        )
        driver.find_element_by_xpath("//button[@id='continue_button']").click()
        driver.find_element_by_id("continue_button").click()
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "os-order-number"))
        )
        assert 'Thank you for your purchase' in driver.title
        print(driver.find_element_by_class_name('os-order-number'))


    @classmethod
    def tearDownClass(cls):
        cls.driver.close()
        cls.driver.quit()

if __name__ == '__main__':
    unittest.main(testRunner=HtmlTestRunner.HTMLTestRunner
    (output='/Users/azad/Desktop/IgavestiWithoutPOM/Reports'))
```

Appendix 2 - Full test suite with POM

Github link of full test suite: `https://github.com/azadnsu/IgavestiWithPOM`

Github link of two login tests: `https://github.com/azadnsu/IgavestiLoginTests`