Faculty of Engineering and Technology

Master of Software Engineering (SWEN)

Master Thesis

Automatic Generation of Selenium Test Cases for Web Applications

Student: Refat Othman: 1175425

Supervised By:

Dr. Samer Zein

Automatic Generation of Selenium Test Cases for Web Applications

By: Refat Othman

Approved by the thesis committee:

____

Dr. Samer Zein, Birzeit University

____

Dr. Adel Taweel, Birzeit University

____

Dr. Radi Jarrar, Birzeit University

____

Date approved:

____

Abstract

Web applications are prevalent and considered the mainstay of information systems for organizations. At the same time, web applications are getting more complex and costly for development and testing. Employees, customers, and business partners rely on these information systems to accomplish their business processes and tasks. Accordingly, users of these web applications assume that these systems are error-free and reliable. The testing aims to make sure the quality of the application works as expected so that the software will be without any bugs. Testing is applied to increase effectiveness, efficiency, and coverage. Automation testing is imperative to assure regression testing, off-load repetitive tasks from test engineers, and keep the pace between test engineers and developers. It can reveal defects to QA engineers or testers at the early development stage when parts of the software are broken or changed. Automated tests save time because automated test cases give the ability to run the cases at night and testers have time to write new tests and automate them. Tool automation help testers automate the test cases and execute them. For web testing, many test cases need a lot of effort, especially time for generating test cases, and there are a lot of studies that present a solution for test case generation. However, we provide a solution for generating test cases for web applications. This research aims to provide and develop a new model-based approach that automatically generates test cases utilizing Domain-Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL) to provide a customizable way for automatically generating test cases. Proof of concept tool was implemented and presented to measure the user acceptance, efficiency, and effectiveness of the approach used to generate code for the tests. MAJD was evaluated using a case study conducted on 20 testers and developers from different experience levels. The approach used to autogenerate selenium code for the tests of the web applications. The results show an efficient tests case generated from the MAJD tool.

Acknowledgments

My thanks and appreciation to Dr. Samer Zein for his effort and time. I'm pleased to work with him also appreciate his help and supportive encouragement throughout the research time. I am grateful to my family, Mom, wife, son, brothers, and sisters for their encouragement and great support.

Table of Contents

Abstract....................3
Chapter 1 Introduction....................9
1.1. Research Problem: 10
1.2. The main objectives: 11
1.3. Main contribution....................11
1.4. Solution Approach 11
1.5. Structure of the thesis....................11
Chapter 2 Background and Related Work 13
2.1. Web application and Software testing....................13
2.2. Model-based Technique....................14
2.3. Visual And Textual Languages (DSVL and DSTL) 14
Related Work....................14
2.4. Search Method 15
2.5. Automated software testing for web application....................15
2.6. Model-Based Testing for web application 17
2.7. DSVL and DSTL 20
2.8. Summary....................22
Chapter 3 Methodology....................24
3.1. Solution Approach 24
3.2. How The Approach Works 25
3.3. Main Components....................26
Chapter 4 Implementation 27
4.1. Implementing DSVL and DSTL 27
4.2. Model transformation Process....................27
To generate test cases we need to use a model to model algorithms: 27
4.2.1. Model-Model transformation....................27
4.2.2. Model-To-Code transformation[24]....................29
4.2.3. Test steps Meta-Model....................30
4.2.4. Test steps model....................30
4.2.5. Test Case Modeling process....................31
4.3. Code Generation algorithms....................31
4.3.1. Custom test generation algorithm 31
4.4. Type of test cases. 32
4.5. Tool's Use Case diagram. 33
4.6. DSVL and DSTL Modeling languages 34
4.6.1. Domain-specific Visual Language (DSVL) 34
4.6.2. Domain-specific Textual language (DSTL) 35
4.7. How to use 36
4.7.1. Generate JSON file for the steps:....................39
4.7.2. Generated Selenium test case code 40
Chapter 5 Experimental design....................41
5.1. Testers Background....................41
5.2. Evaluation Setup 41
5.2.1. MAJD tool on Link git....................41
5.2.2. Website 41
5.3. Environment setup 41
5.4. Evaluation Metrics 42

5.4.1. Tester experience 42
5.4.2. Ease of learning 42
Chapter 6 Results and Discussion 43
6.1. Participants experience 43
6.2. Ease of learning 46
6.3. Usability Questions 49
6.4. Failures, mistakes, and errors 50
6.5. Threats to validity 55
Chapter 7 Conclusion and Future Work 56
7.1. Conclusion 56
7.2. Future Work 56

List of Tables

Table 1 Sample Test Cases for website. 32
Table 2 testers answers for tester's experience questions. 43
Table 3 Participants answers questionnaire's part 2 questions 46
Table 4 Part 2 questions of the questionnaire after each of tester did his task. 48
Table 5 Part 2 questions of the questionnaire. 48
Table 6 - Part 1 interview question 60
Table 7 - Part 2 approach evaluation 62

Table of Figure

Figure 1 TOM Framework....18
Figure 2 Model-Based Testing workflow [10] 19
Figure 3: Model-Based For RAPPT TOOL....20
Figure 4: CDGenerator tool....21
Figure 5 High-level representation for the tool approach....25
Figure 6: Main Components of the MAJD Approach 26
Figure 7 Activity diagram showing how MAJD is used to generate a test cases 28
Figure 8 Class diagram for MAJD tool 28
Figure 9 Overview of the Transformation Process to generate a test case for the web application, workflow for automated test case generation, and test meta-model 29
Figure 10 Login test cases 33
Figure 11 approach use case Diagram 34
Figure 12 Condition DSVL/DSTL 36
Figure 13 Write action by XPATH....36
Figure 14 Write action by ID....37
Figure 15 Write action by Name 37
Figure 16 Read action by Xpath 37
Figure 17 Read action by ID....37
Figure 18 Read action by name 37
Figure 19 Click actions by XPATH....38
Figure 20 Click actions by ID....38
Figure 21 Click actions by Name 38
Figure 22 Condition figure 38
Figure 23 MAJD GUI....39
Figure 24 JSON steps file....39
Figure 25 Selenium code for test case....40
Figure 26 Participants' experience graph 45
Figure 27 Ease of learning graphs 49
Figure 28 Usability questions....50

Chapter 1 Introduction

Web applications and web portals are considered the mainstay of information systems for organizations. Employees, customers, and business partners rely on these information systems to accomplish their business processes and tasks. Accordingly, the users of these web applications assume that these systems are error-free and reliable. During the software development lifecycle, the software testing phase is considered the primary phase to assure the correctness and robustness of the software product. During this phase, test engineers apply different testing methods such as white-box, black-box, unit tests, performance, and usability tests, to mention a few. These testing methods aim to make sure the quality of the application works as expected so that the result will be defect-free software. Quality can be done through software testing [4,7]. Testing check if the software meets all requirements, gives the correct output for the different inputs, completes the tasks and finishes within a short time, and runs the software in different environments [1].

Automation Testing software is applied to increase the test case's efficacy, efficiency, and coverage, thus, freeing test engineers to accomplish important tasks such as exploratory and usability testing. Firstly, test automation is an important aspect of Agile software development methods [5]. Secondly, automation testing provides defects to testers early stage when parts of the software are broken or changed. Thirdly, automated tests save time because automated test cases give the ability to run the cases at night and testers have time to write more test cases. Finally, testing is convenient for a large project and in the repeatedly changing code where regression testing is needed to increase the effectiveness and efficiency of software [3].

Automation tools help testing engineers easily automate the tests and execute them. Open source automation tools have less cost than commercial automation tools. Commercial automation tools benefit the testers with full support, which is not available in open-source tools. Open-source tools have many advantages, like always adding continuous enhancements to the tool. Quick Test Professional (QTP) and Test Complete are commercial automation tools, but Selenium is open-source. Quick Test Professional (QTP) automation tool and Selenium are the most used in automated software testing. QTP is not always preferable related to high license costs. Selenium

is more popular and used by testers. It is possible to write the scripts using many languages such as java, .Net, Perl, PHP, Python, and ruby.
Moreover, Selenium is supported in different platforms such as Windows, MAC, UNIX, and Linux. Selenium seems to be an efficient tool but requires development skills. Modeling techniques, such as Domain-Specific Visual Languages (DSVL), can improve developer efficiency and simplify the design of test cases. DSVL can be applied with Domain-Specific Textual Language (DSTL) to provide a higher abstraction model when designing test cases.

There indeed exist several studies [3][6][10][16][15][18][19] that provide solutions for test case generation. This research aims to introduce a new model-based approach for test case generation that can automatically generate test cases. More specifically, the approach will utilize a new notation and new model to the Domain Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL) to provide a customizable way for automatically generating test cases.

1.1. Research Problem:

Web application testing is very important to ensure the software and the system are error-free and reliable. The testing phase needs more time and cost while the tester applies testing methods such as white-box, black-box, unit, performance, and usability tests. The testing phase makes sure the quality of the software works as expected with defect-free, and meets requirements [1]. Automation testing helps find the defects in the early stage when parts of the software are broken or changed. Automated tests save time because automated test cases give the ability to run the tests at night, and testers have time to write more cases [2][3].

Test engineers need to execute many test cases to cover all user scenarios and software functionalities, and Writing test cases can be time and effort-consuming—the research focuses on assisting test engineers with auto-generation test cases. More specifically, the aim is to extend Model-Based development methods, namely, DSVL (Domain Specific Visual Language) and DSTL (Domain Specific Textual Language), to construct a framework that can auto-generate test cases. Our approach that automatically generates the test cases needs to add a new notation and

model to the Domain Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL) to provide a customizable way for automatically generate test cases [21][22].

1.2. The main objectives:

1. Extend for web Domain-specific Visual/Textual language (DSVL/DSTL) to enable test case presentation.
2. Develop a model-based framework that automatically generates test cases based on DSVL and DSTL notation.
3. We evaluated our approach using a focus group case study that measures the efficiency, user acceptance, effectiveness, and usability of the model.

1.3. Main contribution

The main contribution of this thesis is helping testers and developers to generate test cases for the web application. The tester only needs to provide the steps by using a user interface that leverages the Domain Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL), allowing the testers to design the test step and automatically generate the code of the tests.

1.4. Solution Approach

The main idea for this thesis is to provide a model-based approach that automatically generates code for test cases, which leverages the Domain Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL) to provide a customizable way for automatically generating test cases for the web application. This approach aims to assist and help developers/testers who do not have automation skills. The evaluation of this tool was done by asking participants with different levels of experience and skills to measure user acceptance efficiency and effectiveness.

1.5. Structure of the thesis

The structure of this research contains a lot of chapters, and it consists of the following:

- Chapter 1: Covers an introduction about the research problem, motivation, and research questions.
- Chapter 2: The background of the auto-generated test cases. Web application, Model-based testing, and Domain-Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL) to provide a customizable way for automatically generated test cases.
- Chapter 3: Contains the related work and literature review for related works, including the group of related works. It also summarizes the literature review of the related work.
- Chapter 4: Introduces the research methodology, describes and presents the research approach of our framework, and evaluation of the approach.
- Chapter 5: Describes the solution approach implementation details, implementing the tool's architecture, generated code architecture, tool's design, and using examples.
- Chapter 6: Describes the experimental design details using examples.
- Chapter 7: Presents the evaluation results and discussion of the implemented approach evaluation, as well as the possible threats to validity
- Chapter 8: Concludes this thesis and provides avenues for future work.

Chapter 2 Background and Related Work

2.1. Web application and Software testing

All companies today rely on web applications for all operations every day. Therefore, the data of web applications and the web application's functionality should be without any errors. The system needs to check the quality and the functionality of the application works as expected. The web application runs on a central high-performance device called "Server," and clients can connect to the server using relatively low-performance devices and request the application services. Web applications have evolved from limited, static, and simple to interactive, multi-services, and complicated applications. They are developed using web technologies such as HTML, CSS, JavaScript and can be accessed using any preferred web browser such as Chrome, Opera, Firefox, and IE [6].

These days, employees, customers, and business partners use web applications to do their business processes and tasks. Accordingly, the users of these web applications assume that these systems are error-free and reliable. Web Testing can be done to check if the software meets all requirements, gives the correct output for the different inputs, completes the tasks, finishes testing within a short time, and runs in different environments [1]. Automation testing is widely known, and many companies use it due to its advantages in reducing cycle time, which helps an engineer decrease testing costs in the software. It shows and saves a lot of benefits such as lowering the budgets and increasing the quality. It is also considered a core component in agile development [6]. People need certified applications that employ the internet as an essential tool for their business. Web applications are developed to meet this need and provide a way for people to communicate and collaborate to achieve their business goals efficiently and quickly.

Software testing is performed to find out how well an application works and find errors in the system. The testing aims to make sure the quality of the application works as expected so that the result will be defect-free software. The quality of any software can only be known through software testing [7,4]. Testing can be done to check if the software meets all requirements, gives the correct output for the different inputs, completes the tasks, finishes testing within a short time, and runs in different environments [8].

2.2. Model-based Technique

Model-based techniques abstract the details of the software development. They show the development process. The most important thing is that it will improve the developers' productivity by using Model-based techniques to finish all tasks easily without exhausting the details of the developments [21]. It is also a black-box testing technique used to compare the behavior of the implemented system. MBT is popular in the automation GUI testing field, not limited to mobile app testing only, but also with other software platforms. TOM [15], for example, is a model-based testing framework that automatically generates user behavior test cases for web applications. In addition, Baek, Y.-M et al. [23] support the effectiveness of model-based testing by using MBT with multilevel GUICC (GUI Comparison Criteria), which achieved higher effectiveness than other testing approaches in terms of code coverage and error detection ability when it was evaluated using empirical experiments.

2.3. Visual And Textual Languages (DSVL and DSTL)

The visual and textual languages (DSVL and DSTL) that represent and give the details of the component, both the DSVL and DSTL, will be constructed and evaluated to ensure that they can model test cases for web applications in selenium that add a new notation to use it in our approach to generating the code. Domain-specific Visual Language (DSVL) consists of GUI elements representing a concept in an automated test case, and these elements or notations correspond to the test case model. At the same time, Domain-specific Textual language (DSTL) consists of a set of textual notations used by a tester to add or edit test cases [21,22].

Related Work

This literature review will focus on generating test cases related to model-based automation studies, the framework for test case generation for web applications, and visual and textual languages(DSVL and DSTL) techniques. Google Scholar, IEEExplore, Springer, and ACM searched for related studies.

In this chapter, we present a comprehensive review of the selected studies (categories) that have approached web application testing, web application test case generation, automated testing for web applications using tools such as selenium, and domain-specific Textual/Visual language.

2.4. Search Method

The following search methods:

1- Number of pages: we select the paper that has at least 5 pages.

2- Type of paper: we select the papers that are empirical to check the results and compare the papers based on the results.

3- Publish year: we collected the papers from recent years because we have many papers that talked about testing web applications and test case generation

4- Keywords: These criteria we are using many keywords such as:

   1. Web application testing
   2. Software testing
   3. Web testing
   4. Test case generation
   5. Model-based test case
   6. Automatic test case generation
   7. Model-based testing
   8. DSVL
   9. DSTL
   10. Domain-Specific Languages

5- Database: we used the ACM, IEEE database, and Google Scholar to collect these papers.

2.5. Automated software testing for web application

These days, all companies use automation due to advantages such as no need for a human to check the functionality, reducing cost and cycle time for any software. Automation testing shows and increases the quality of the software and tests the cases in effective ways [11]. Framework for automated testing to provide high quality to increase the quality of the application using software to control test script to check the steps executed correctly by checking the actual results with expected results. Automation testing has many advantages to software testing. The efficiency of the software for time to execute and test using automation is less than executing the tests manually. Another thing repeatability is the same test can be executed many times without

any action from the human. When creating a framework, it should have some properties, such as while the test needs to do anything like any action or anything, it will do it with a call function or method from the framework. Another thing is while the framework built the tests, it should be simple, not complicated because complicated needs time for maintenance. The last one is the framework should test the tests and execute the code of the tests [12].

Many companies use automation testing due to its advantages for reducing cycle time, helping the engineer decrease testing costs in the software. It shows and saves a lot of advantages such as decreasing the budgets and increasing the quality. It is also considered a core component in agile development [6].

Furthermore, software testing tools are divided into different categories:

- Testing management tools used for storing information.
- Load testing tools are used to determine the behavior under different loads.
- Testing functional tools.

Testing tools are useful to record, play, and re-execute test cases repeatedly. Many powerful tools are used for automated test execution, such as Selenium and Testdroid [9][14]. Selenium is the most common tool used for mobile applications and web development [9]. Automation can lead to many benefits, such as higher software quality and cost savings. Also, it can manage time and cost and improve the process effectiveness by reducing the risk of human error, making tests more repeatable, and improving the process efficiency. It depends on stability and structure in the testing and successful processes. The replacement of manual work by automation will cause a major change in the tester's daily work. So changes to the work require major training, and training requires both a budget and time and priority in the work schedule [10].

An example of a framework for automation testing a web application is "Software automating testing" (SAT). This framework reduces the time needed for testing and reduces the cost of previous articles. In contrast, this framework focuses on the performance of the framework kike test script creation time to automate and generate the complete test reducing around 68% of the total time for automating the tests. The other way to check the framework performance is by using and checking if the tool provides usability by using keywords on SAT. This supports adding new keywords by testers using the SAT framework. Maintainability: The framework generates the code for the test, and the testers can edit and check the code of steps. The code of

the steps is no need to have a development skill because the test and framework use an automation tool [13].

Another framework for automation testing for a web application called jFAT integrates with selenium and TestNG. The integration with selenium and testNG will also create an efficient and clear report for the test results, and it will allow the testes to easily automate the tests scripts. Framework tested by using the application for banking transactions for managers, such as adding, updating, removing the customer from the application and system, depositing or withdrawing money from accounts, and checking the balance for any customers [14].

2.6. Model-Based Testing for web application

MBT is the most popular technique used for automation UI testing, and TOM is a model-based testing technique that automatically generates the test cases for the web application. Model-based for TOM comes to solve the problem related to selecting the user's perspective test cases by using MBT for generating test cases for the relevant users. Figure 1 below shows the MBT framework for TOM, consisting of an adapter layer and a core layer. The Adapter layer is an interface that connects the framework's core to the test automation framework. It is consists of and includes a Model that imports the model for UI and test cases exporter that needed to generate test cases. The second layer shows the graph model representing user action on the interface, such as action buttons on web pages. This layer has a path generator, test case generator, and concrete test cases that provide effective mutation test cases from the graph [15].

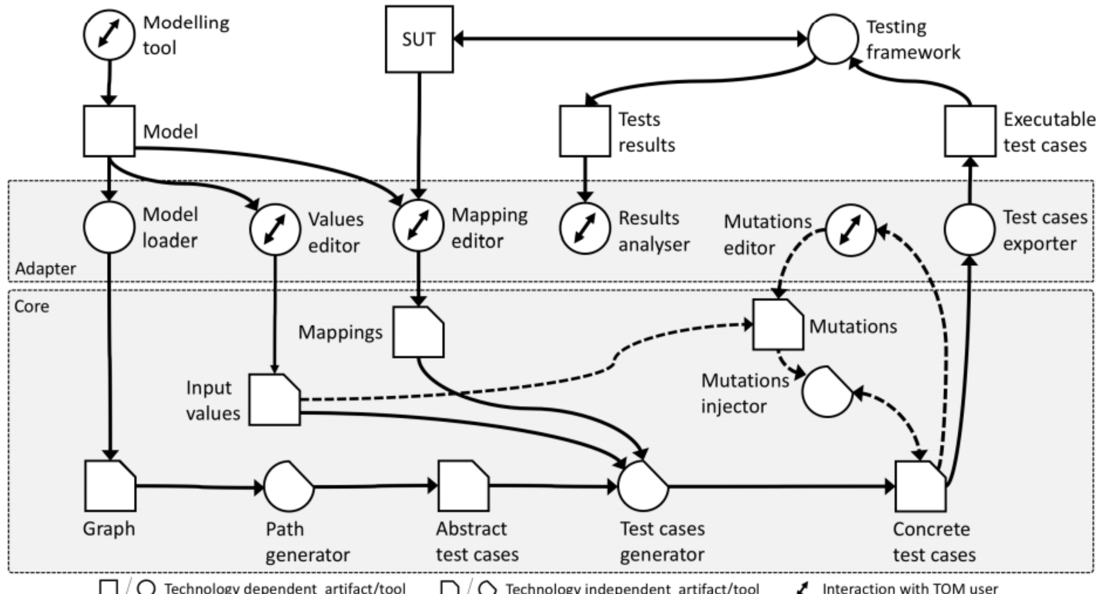


Figure 1 TOM Framework

TOM evaluated using a real website called OntoWorks. The TOM framework used to generate the testing system model ends with 15 states and 24 transitions, and the home page, for example, had 61 validation checks. The evaluators focused on the three main user-defined mutations for web applications: web page refresh, back button click, and double-click UI element. Finally, the total number of paths was 273, with 2,730 generated test cases. The results show 935 test failures appeared while testing OntoWorks. These results show an implicit implementation problem of the tested webpage. It used the same identifier many times, which should be unique for any element within the page. It also provided a few aspects for the researcher to improve TOM, as they mentioned. Model-based testing generates test cases and runs all the tests cases [16]. Model-based are consist of as follows:

1. Create models from the specification and requirements of the application
2. Generate test cases for the application. In these steps, generate the cases from models created from step1.
3. Concretize cases, execute the test cases that do not have implementation details and can be run and executed directly.
4. Execute the test cases, execution the cases manually, or using an automated test execution for each test should be test passed or failed
5. Report results. The user should see the report about the test cases if it's passed or failed

The paper supports and provides a tool used to test case generation from Model-based testing models and deliver a comparison of model-based testing tools [16]. While [17] presents an approach for web application testing that is constructed and checked to trace the specification and requirements using the JTSG algorithm that generates the scenario of the test cases to build it and generated for any web application given.

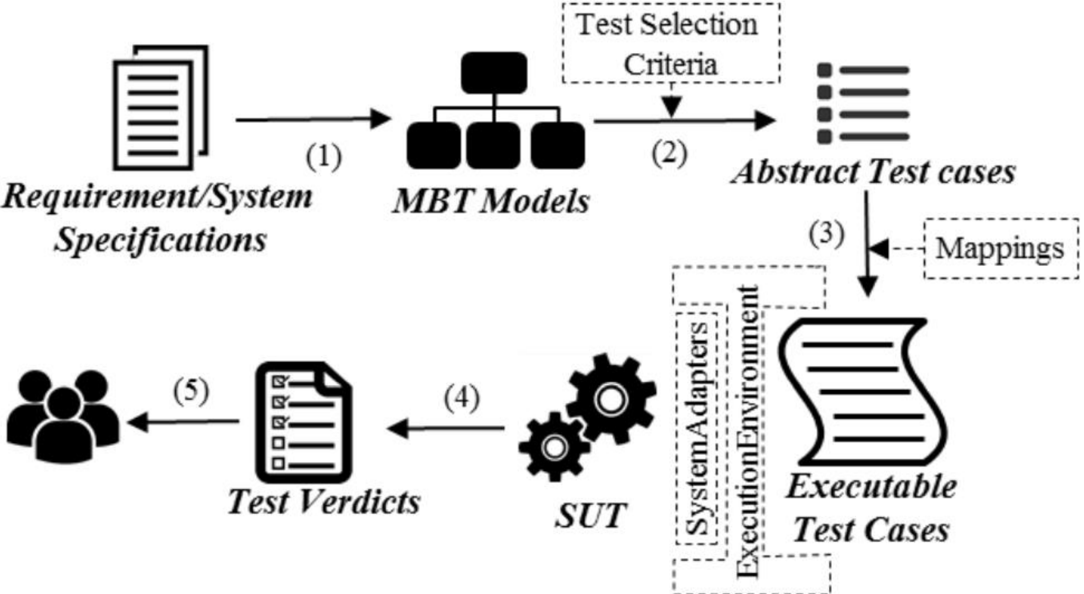


Figure 2 Model-Based Testing workflow [10]

An approach to generate test cases that used the mutation operators add a step, remove, repeat, swap, insert and add back steps. By using this, operators can generate new test cases based on the

existing test cases to test the application with less effort and increase the automation for a web application [18][19].

Testing is the time and cost that the tester needs to check the quality of the software. The main idea for testing is to support and develop an application without any defects and ensure after-code changes do not produce any defects. N. Gupta et al. [20] provide techniques and approaches used for test case generation for web applications. They select an automated tool for test case generation according to the different scenarios on a web application to reduce the efforts and cost.

2.7. DSVL and DSTL

In [21], the author designed a new approach or tool called RAPPT stands for Rapid Application Tool, which helps the developers to understand the application that provides many views of the application by using Domain-specific visual language and Domain-specific textual language techniques to help the developer to define and develop the application using notation (DSVL/DSTL) to have an abstract view like page navigation. These views help the developer to understand, develop, and enhance the application. On the other hand, the tool presents mockups about the application to enhance communication between stakeholders.

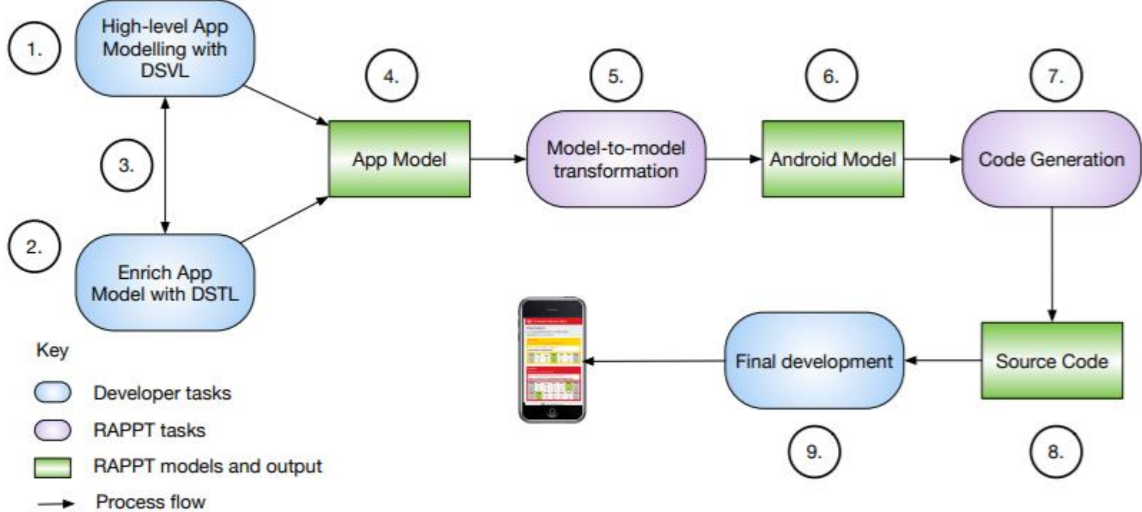


Figure 3: Model-Based For RAPPT TOOL

The developer starts to describe the high-level structure of the application and the number of screens in the application by using DSVL. The developer adds and provides more details to the application by using DTSL like configuration authentication and info displayed in GUI. Then uses a model to the model transformation that has the details and the information needed to generate the code to ensure the generated code is almost closed to the code that the developer will write—after that, it uses a template to generate the code. In addition, developers need to add styling and business logic to have a good user experience and accept the application. The authors evaluated the approach using a user study with 20 participants (17 male, three female) with different experiences. First, they conducted a demographic question that talked about the backgrounds of the participants. Then they introduced a video about RAPPT that helped participants understand the tool and how it works. After that, they asked the participant to fill out the new questionnaire, which gave feedback about the tool. The results conclude and show the acceptance of the tool.

In [22], the author designed a new approach or tool called the CDGenerator tool to help the developers and assist those who do not have computing skills. It is persisting their application data locally. The approach using Model-to-Model and Model to code techniques also uses Domain-specific visual language and Domain-specific textual language techniques to create data persistence and provide a customized way to generate data automatically.

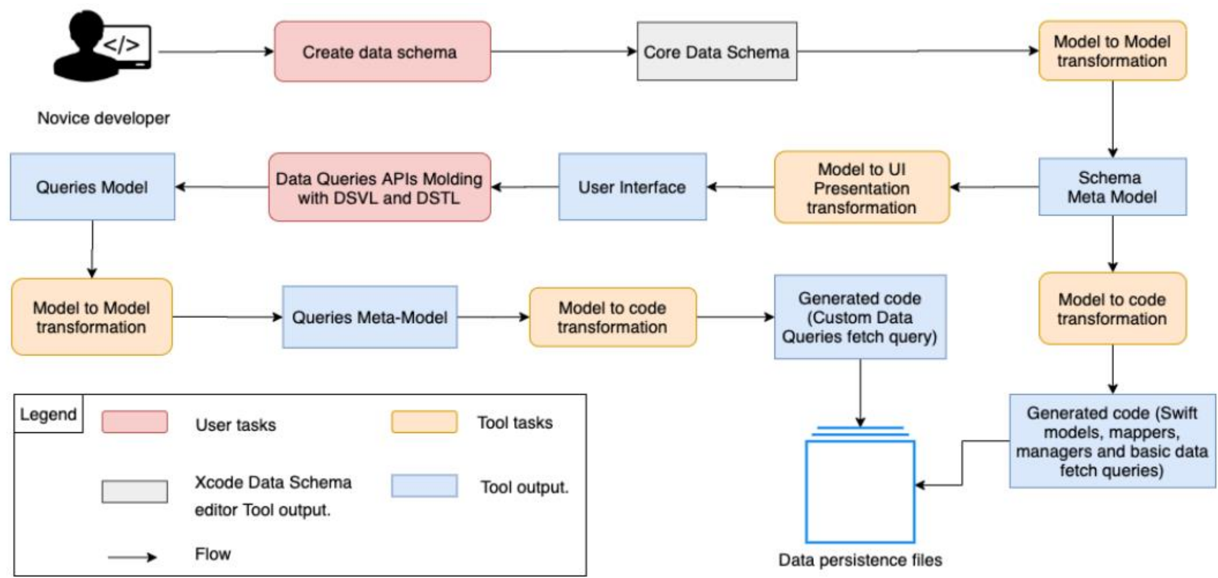


Figure 4: CDGenerator tool

The tool's implementation consists of two main steps. The first is generating data persistence components based on data schema. The second is using DSVL/DSTL to create data fetch queries. The first step needs the developer to attach the schema file to the CDGenerator tool to generate the data classes. The tool reads the schema file and generates a meta-model that represents the data schema containing all the information about the relationships between entities. Then generated code or generated domain is used to generate data for the application. The developer can use DSVL/DSTL to generate data fetch queries in the second step. After the schema is generated in step 1, apply a GUI model to provide a good GUI that must present the data schema to have a simple, usable design. The developer uses the UI to specify the query's data that needs to generate. Also, they can see the data schema and change the properties or specify the function or condition to be applied. Also, a developer can add a notation to the query by using DSVL/DSTL to generate query data that contain all info or data related to the query for generating the code. After that query metal model is used to generate the data query and display the code in simple UI. The developer can easily edit or copy the query and use it in his application. The authors evaluated the approach by using a user study with 6 participants (3 male, three female) with different experience levels. First, they conducted a demographic question that talked about the experience and backgrounds of the participants. Then they introduced a video tutorial that helped participants understand the CDGenerator tool and how it works. After that, they asked the participant again to use the approach and fill out the new questionnaire, giving feedback about the CDGenerator tool. The results conclude and show the acceptance of the CDGenerator tool.

This thesis's solution approach benefits from textual and visual modeling techniques to provide a highly efficient modeling approach that helps testers generate test cases.

2.8. Summary

This chapter provides a comprehensive picture of software testing, especially automated test case generation, and discusses it through the literature review to better understand the concepts. Many research papers have been published related to testing case generation, and these papers present knowledge about the steps and model-based for test case generation. These papers also show the advantages and disadvantages of applying all testing approaches and methods.

Despite the increase in the number of achievements related to generating test cases and the increase in the number of research related to it this thesis, we aim to extend Model-Based development methods, namely, DSVL(Domain Specific Visual Language) and DSTL (Domain Specific Textual Language), to construct a framework that can auto-generate test cases in order to help testers to generate tests for web applications without writing any code and without requires any automation skills.

Chapter 3 Methodology

The main goal of this thesis is to help the testers and developers automatically generate tests cases for web applications using the model to model transformation and model to code transformations for models that specified using Domain-Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL) which was presented by Barnett et al. [21][22]. This thesis aims to develop a tool that has these concepts that will assist testers and developers in generating test cases and executing them.

This chapter will consider how we can use the tool, solution implementation, and evaluation of the tool.

3.1. Solution Approach

Figure 5 below shows the representation of the approach. The tester needs on step 1 to describe the high level of the test case using DSVL, including the number of steps in the test case and the order of the test case steps. After that, the developer in step 2 can switch to using DSTL to provide additional details for the test case that are not provided by using DSVL, like condition code that compares values. Both DSVL and DSTL update the test case model allowing the tester to switch between the interfaces as they proceed through the modeling process. Testers can view the code that will be generated from their model. The approach performs on step 4, a model to model transformation to convert the test step meta-model to a test step model. The test meta-model contains all the required information needed to generate the code for the test cases steps. Step 6 will map code templates to the test model and a source code for the test. In step 7, the generated code for the test case from the tool contains the code for the test cases that is ready to be run and executed.

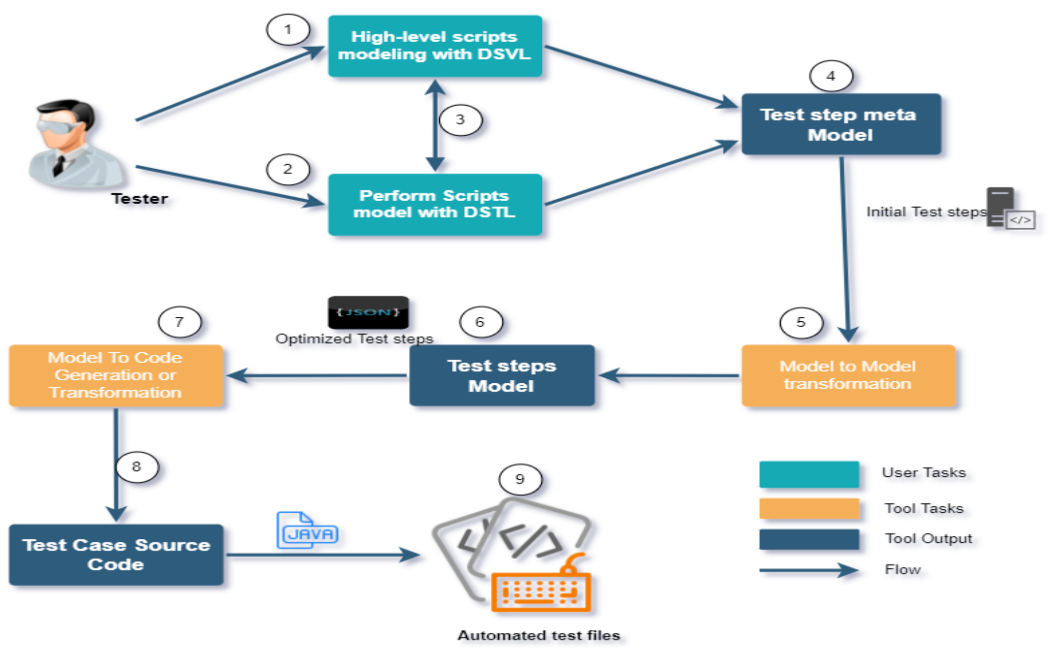


Figure 5 High-level representation for the tool approach

3.2. How The Approach Works

This section describes details how testers or developers generate test cases for web applications, as shown above in figure 5.

The implemented tool works in the main step called generating test cases for web applications using Domain-specific visual and textual modeling language DSVL and DSTL. The testers can generate test cases by specifying its details using visual and textual modeling notations DSVL and DSTL by doing the following steps:

1. Participants run the tool
2. Once the tool GUI appears, this GUI represents all notations that be used from participants to generate test cases. In a simple, easily usable way, the participants can easily use it to add, delete and edit steps of the test.
3. Participants use the GUI to specify the steps they want to generate. They can view the test steps specify methods and functions to be applied or conditions. The participant specifies his test by selecting relevant GUI elements that represent the step specification, and the participant also can edit, edit and delete extra-textual notations to add it for generating the test. The participants can select the action for the step by choosing notations to do that action.

4. Once the participant finishes adding his/ser specification to the test, he needs to execute generate Json button test meta-model using model-to-model transformation (MTM). The test generates all steps related to the test for code generation.
5. The test meta-model is then used to automatically generate the test and display its code to the participants.
6. The participant can easily run and execute the code for the test generated.
7. simply by repeating steps 1-6, the participants can generate more tests.

3.3. Main Components

Figure 6 below shows the main components of the solution approach tool. Test customizations User interface, which provides participants the ability to customize their step of a test using DSVL and DSTL modeling, Model to code generator used to generate selenium code for the test steps.

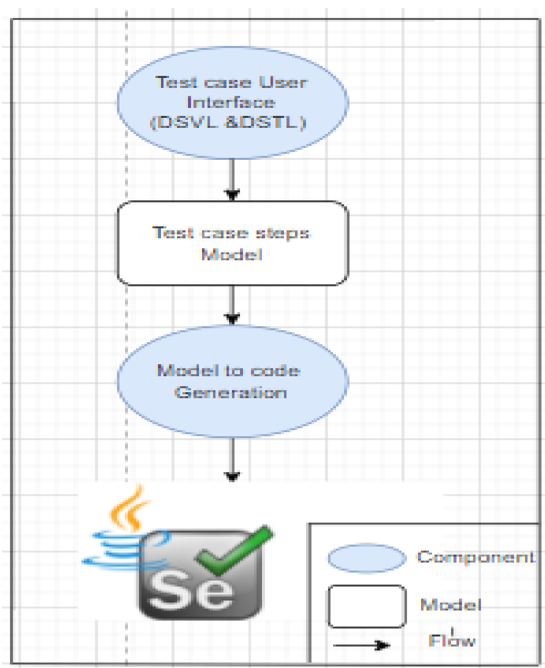


Figure 6: Main Components of the MAJD Approach

Chapter 4 Implementation

This chapter will present the implementation details design of the presented approach. That was implemented as a proof-of-concept tool called MAJD.

4.1. Implementing DSVL and DSTL

The visual and textual languages (DSVL and DSTL) that represent and give the details of the component, both the DSVL and DSTL model, will be constructed and evaluated to ensure that they can model test cases for web applications in selenium. After that, add a new notation to use in our approach to generating the code. Domain-specific Visual Language (DSVL) consists of GUI elements representing a concept in the automated test case. These elements or notations correspond to the test case model, while Domain-specific Textual language (DSTL) consists of a set of textual notations used by a tester to add or edit test cases. Section 4.5 contains all notations that are used in our approach.

4.2. Model transformation Process

To generate test cases we need to use a model to model algorithms:

- **Model-Model transformation**

This process ensures the existence of all needed information to generate the test steps. It transforms the Test Meta-model, which contains all Test steps specifications specified by the user, to a Test model, which contains all information needed to generate test case code and will be the input to the Model-To-Code transformation process, which are also shown in figure 9.

Input: Test steps Meta-Model

Output: Test steps model

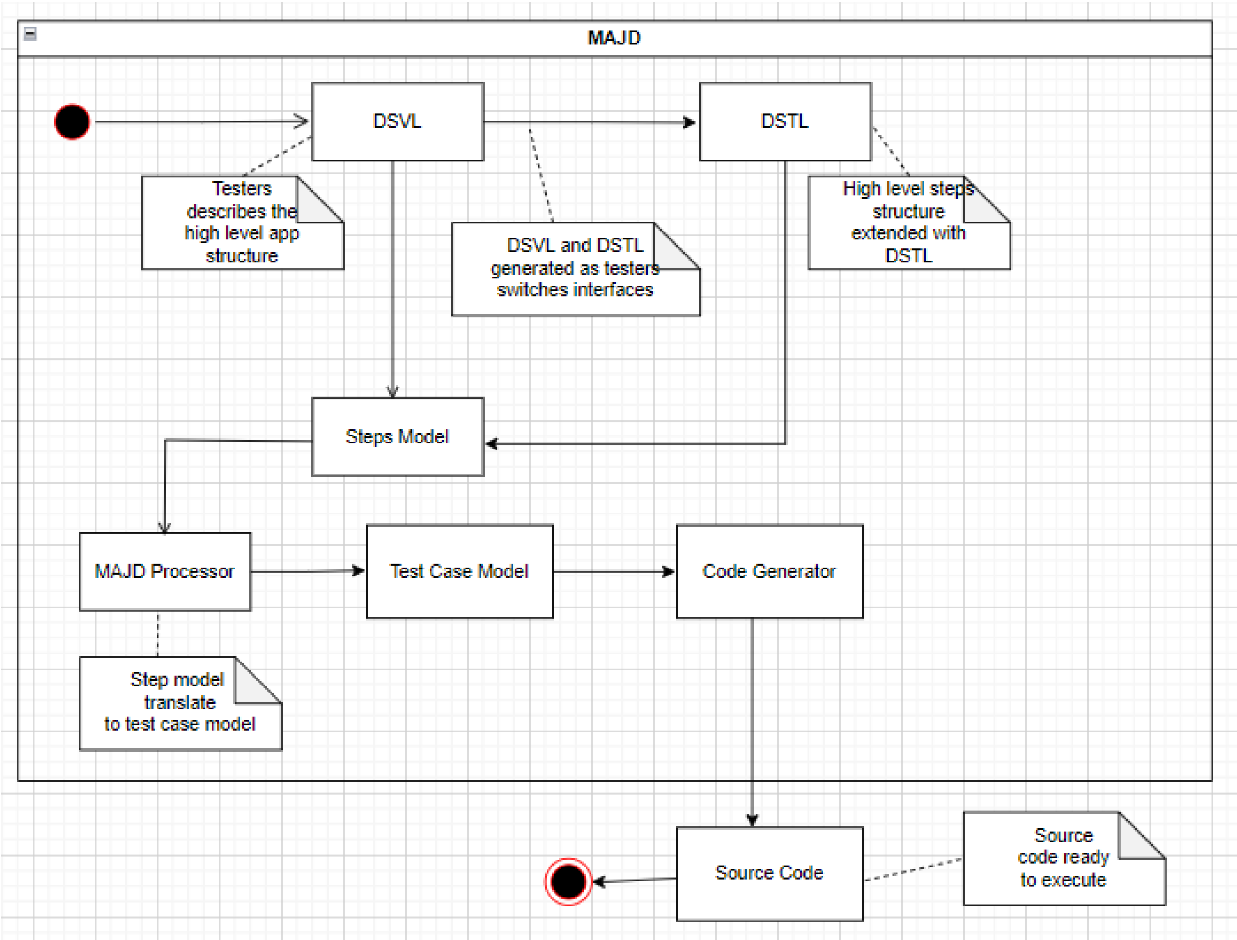


Figure 7 Activity diagram showing how MAJD is used to generate a test cases

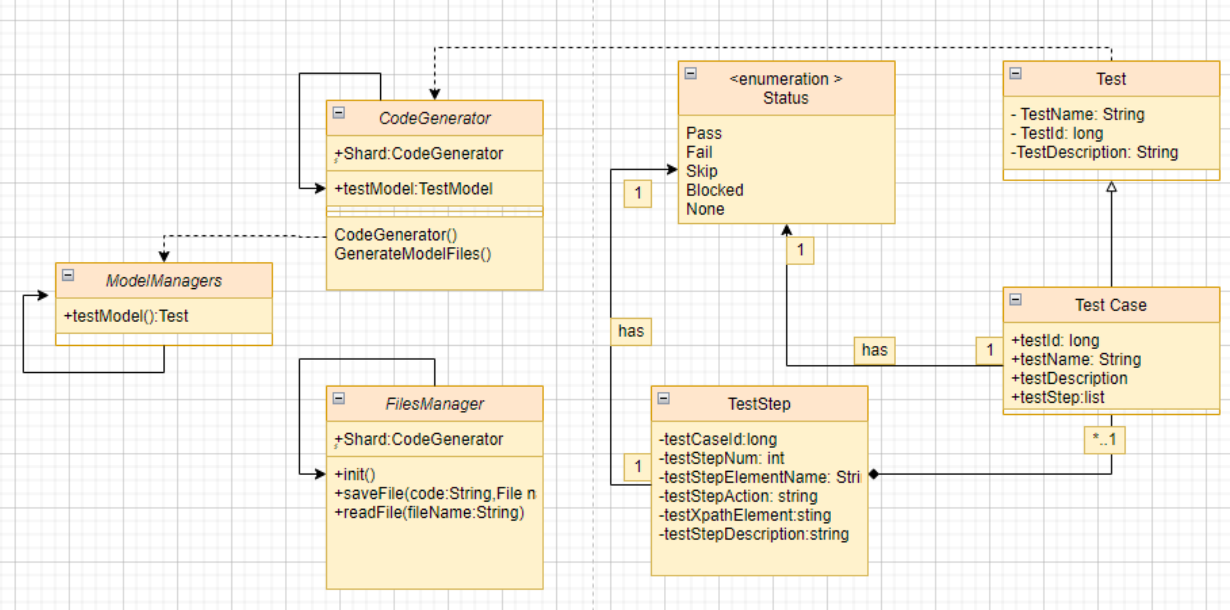


Figure 8 Class diagram for MAJD tool

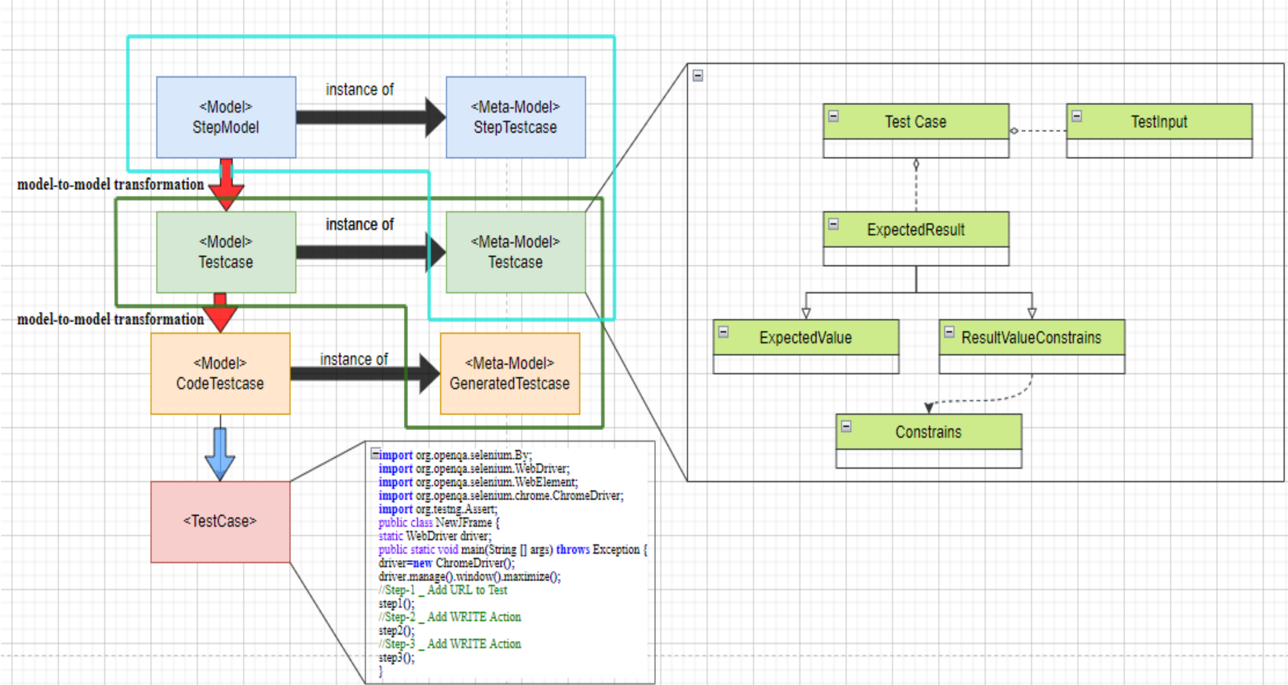


Figure 9 Overview of the Transformation Process to generate a test case for the web application, workflow for automated test case generation, and test meta-model

- **Model-To-Code transformation[24]**

This algorithm transforms the tests steps model to the test case source code. The generated code will be ready to be executed and check the results.

Input: Test steps model (JSON)


```
[
{
"step#:1":{
"Element Name:":"URL ",
"Step Action:":"URL",
"Step number :":1,
"XPATH Element:":"http:\/\/qaautomationdsvl.000webhostapp.com\/index.php",
"Step discription:":"Step-1 _ Add URL to Test",

}
},
{
"step#:2":
{
"Element Name:":"admin",
"Step Action:":"WRITE",
"Step number :":2,
"XPATH Element:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[1]",
"Step discription:":"Step-2 _ Add WRITE Action ",
"action Type:":"xpath"}
},
{
"step#:3":
{
"Element Name:":"admin",
"Step Action:":"WRITE",
"Step number :":3,
"XPATH Element:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[2]",
"Step discription:":"Step-3 _ Add WRITE Action ",
"action Type:":"xpath"
}
```

```
}
]
```

Output: Code for test cases

```
/**
 *
 * @author Generated By Automation Tool For QA
 */

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.Assert;


public class NewJFrame {
  static WebDriver driver;
  public static void main(String [] args) throws Exception {
     System.setProperty("webdriver.chrome.driver", "C:\\Users\\admin\\Downloads\\chromedriver_win32
(1)\\chromedriver.exe");
     driver=new ChromeDriver();
     driver.manage().window().maximize();
     //Step-1 _ Add URL to Test
     step1();

     //Step-2 _ Add WRITE Action
     step2();

     //Step-3 _ Add WRITE Action
     step3();

  }
```

The input/output models, their rules, and specifications are presented in the following list, shown in the class diagram in figure 7, figure 8, and figure 9.

- **Test steps Meta-Model**

The test steps meta-model represents the test case steps specified by the developer using DSVL and DSTL. This model consists of all specifications and attributes that describe the test case generated.

- **Test steps model**

The test steps model represents the test model in terms of code, and it contains all the required information about steps needed to generate a test case code.

- **Test Case Modeling process**

The provided steps test case is modeled into a data test case meta-model using model-to-model transformation. This process ensures that the provided data for a test case is valid and prepares it as an input for the DSVL/DSTL modeling by adding all information needed for code generation and UI notification.

Input: JSON test data


```
"step#:2":
{
"Element Name:":"admin",
"Step Action:":"WRITE",
"Step number :":2,
 "XPATH
ELement:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[1]",
"Step discription:":"Step-2 _ Add WRITE Action ",
"action Type:":"xpath"}
},
```


Output: Test Case Model

4.3. Code Generation algorithms

This section addressed the implemented algorithm of code generation it a custom test generation algorithm

- **Custom test generation algorithm**

The proposed custom test case generation algorithm can be described in the following pseudo-code:

Algorithm input: the test steps specified by using DSVL and DSTL

Algorithm Output: selenium test case code

```
public class NewJFrame {
   static WebDriver driver;
   public static void main(String [] args) throws Exception {
        System.setProperty("webdriver.chrome.driver",
"C:\\Users\\admin\\Downloads\\chromedriver_win32
(1)\\chromedriver.exe");
        driver=new ChromeDriver();
        driver.manage().window().maximize();
        //Step-1 _ Add URL to Test
        step1();

        //Step-2 _ Add WRITE Action
        step2();

        //Step-3 _ Add WRITE Action

   step3();
```

```
}
```

The algorithm was applied using the following main steps:

1. Add test steps by using DSVL and DSTL.
2. Transform Test steps meta-model to test steps model by using a model to model transformation.
3. Generate test cases by using the following:
   i. A load JSON file that has all steps
   ii. Generate test steps code by adding each step on each function
   iii. Display generated code for the test cases
4. Execute the tests case that has already been generated by the tool.

The steps performed by MAJD to generate a test case are shown in figure 7. Our tool used DSVL as the first view for generating a test case. Testers can specify the steps of the test. MAJD generates DSTL that testers can edit/add details about the steps. MAJD keeps the two views, the DSVL and DSTL, synchronized so testers can switch between views. MAJD updates the test step model when the tester updates DSVL/DSTL as shown in figure 7. The step model was implemented as a JSON Object. Using JSON as a model format meant that it could easily be sent to another model that used JSON as a data transmission format is shown below.

```
{
"step#:1":{
"Element Name:":"URL ",
"Step Action:":"URL",
"Step number :":1,
            "XPATH
            Element:":"http:\/\/qaautomationdsvl.000webhostapp.co
            m\/index.php",
"Step discription:":"Step-1 _ Add URL to Test",

}
```

4.4. Type of test cases.

Our approach covers all functionality test cases (UI test cases) except tests that need a loop in the code. This type of black-box testing can provide if the app works as expected, and testers identify the software functionality as expected to perform are failure or success. Table 1, figure 10 shows a sample test case tested using our website.

Table 1 Sample login test case.

Test Case ID	1111
Test cases	Test if user is able to login successfully.
Priority	A
Preconditions	User must be registered already
Input test data	correct username,correct password
Steps to be executed	1. Enter input(correct)username and password on the respective fields

	2. click submit/login
Expected results	User must successfully login to the web page
Actual results	
Pass/fail	

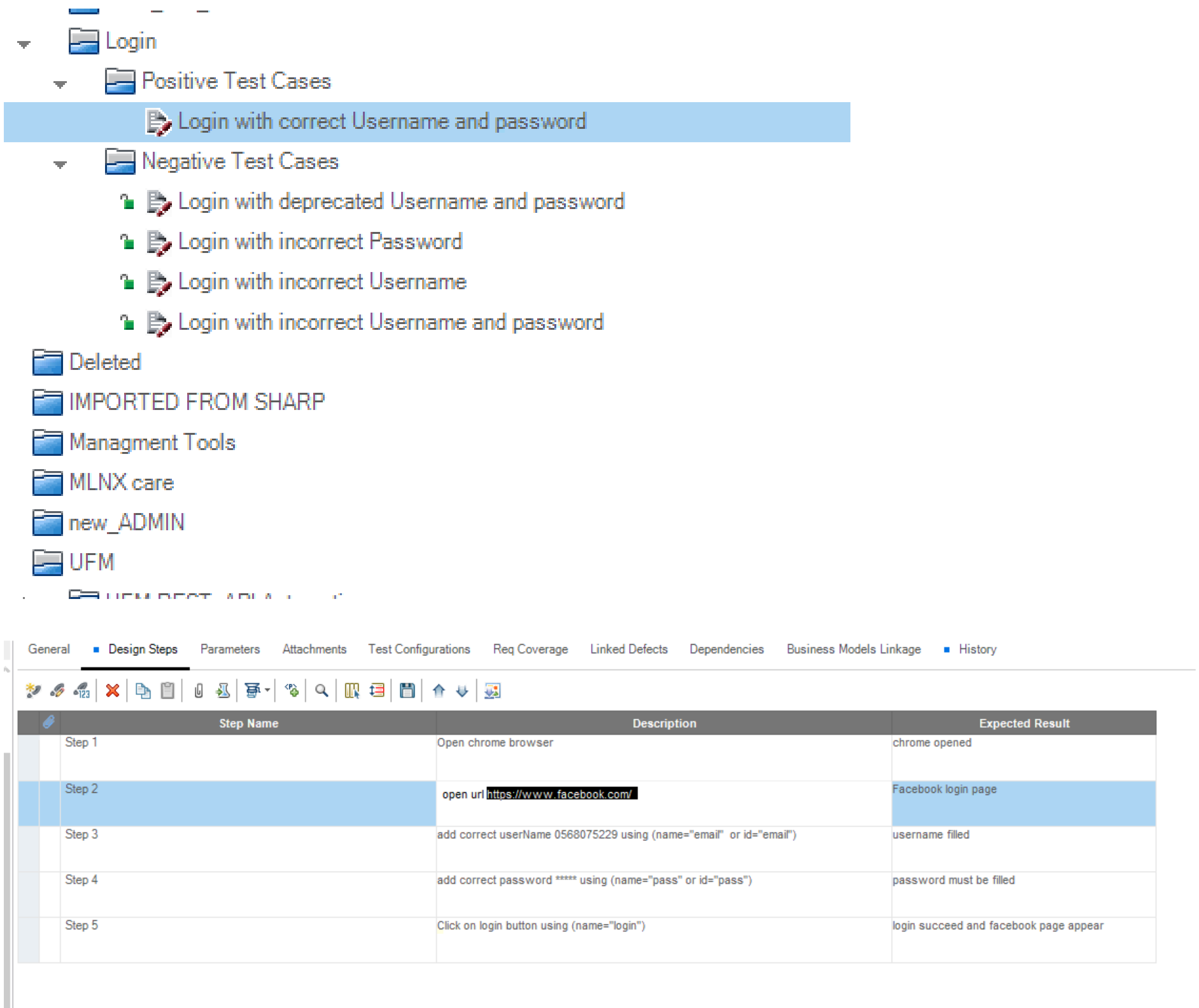


Figure 10 Login test cases

4.5. Tool's Use Case diagram.

Figure 11 shows the use case diagram of an implemented tool, which represents the main functionality of the MAJD tool. Users can generate a test case code by specifying its steps using DSVL and DSTL.

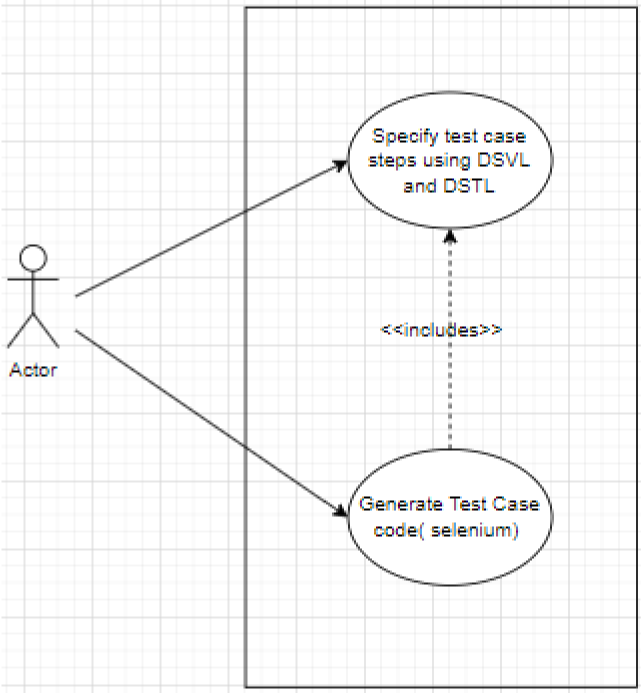


Figure 11 approach use case Diagram

4.6. DSVL and DSTL Modeling languages

This section presents the design of the domain-specific visual language (DSVL) and domain-specific textual language (DSTL), which have been designed based on Barnett et al. [21] DSVL and DSTL modeling languages.

The DSVL & DSTL are visual/textual languages representing and abstracting the detailed test case steps. Participants can use them to specify the details of a test case steps using a relative visual or textual notation.

- **Domain-specific Visual Language (DSVL)**

DSVL contains all UI visual elements and components, and each one represents a specific concept in the test case. Element and components called notations correspond to the test case meta-model that acts as the base of the test step model.

Concept	Notation	Description
Write	</> XPath	Write to any elements that exist on any web page by using XPATH
Write	<Name="?">	Write to any elements that exist on any web page by using NAME
Write	ID	Write to any elements that exist on any web page by using ID

Read	</> XPath	Read to any elements that exist on any web page by using XPATH
Read	<Name="?">	Read to any elements that exist on any web page by using NAME
Read	ID	Read to any elements that exist on any web page by using ID
Click	</> XPath	Click to any elements that exist on any web page by using XPATH
Click	<Name=?>	Click to any elements that exist on any web page by using Name
Click	ID	Click to any elements that exist on any web page by using ID
Add URL	http://	Add URL of the application that need to generate test case
Conditions	Condition ?	Compare code to compare the value with And DSTL Condition for comparing code
Sleep	Waiting!	This notation for set asleep to wait some of the action finished, it's kind of waiting for the loading.

Table5-6 Custom test case visual language

- **Domain-specific Textual language (DSTL)**

The domain-specific visual language consists of a set of textual notations, DSTL notation used by testers for adding or editing specific test steps. We have DSTL notation shown in the table about table 5-6 for comparison. The MAJD DSTL is designed to use the same notation used in web development, so anyone needs to use the MAJD; he/she does not need to learn extra notations. At the same time, the participants who do not know or have knowledge of the notation

can still specify their test case step details using DSVL, such as the below figure 12. [Contains, Begins with, Ends with, regex, etc.]

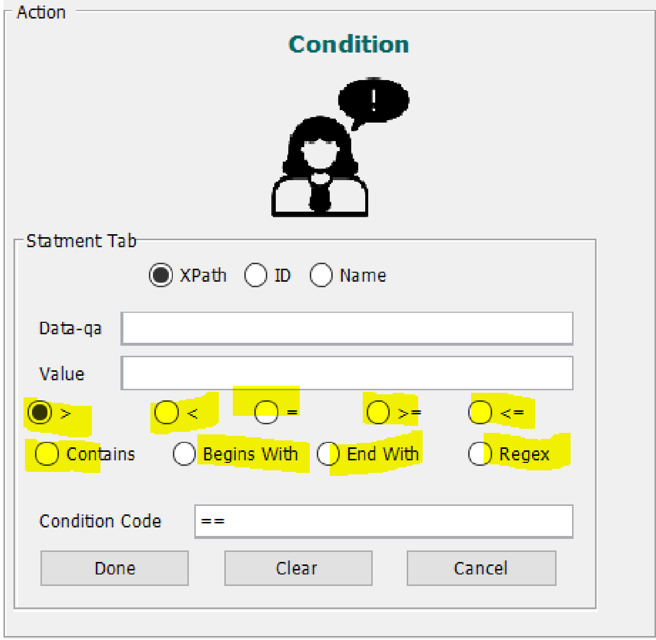


Figure 12 Condition DSVL/DSTL

4.7. How to use

This section presents an example of using the MAJD to generate a test case for a web application by using DSVL and DSTL, and you can see the video here: Video link[1]

To generate a test case for a web application by specifying its specifications using DSVL and DSTL in the following steps:

1. Open GUI to view all notations that appear in figure 23 below.
2. Select add URL notation to open the application that needs to generate the test case
3. Add a set of actions that represent the steps of the tests:
    a. Write action: write action to write on any elements on the web page, such as text fields. The participant needs to add the id or name or XPATH and the input data that need to write it on the element.
        i. XPATH:

Figure 13 Write action by XPATH

        ii. ID

[1] https://drive.google.com/file/d/1Dg4_fdb1D6i6KK5kpWIyp3m1ezbTzTuL/view?usp=sharing

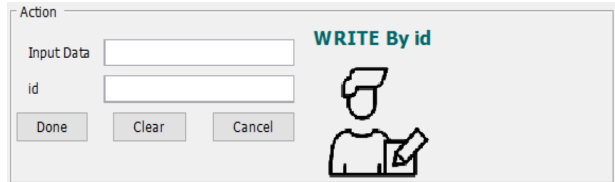


Figure 14 Write action by ID

iii. Name:

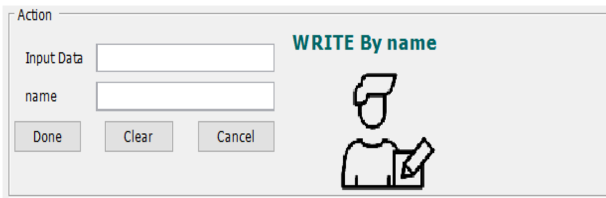


Figure 15 Write action by Name

b. Read action: using Read action to Read from elements that exist on the web page such that labels.

i. XPATH:

Figure 16 Read action by Xpath

ii. ID

Figure 17 Read action by ID

iii. Name:

Figure 18 Read action by name

c. Submit action: using click action to click on elements that exist on the web page such that buttons.

i. XPATH:

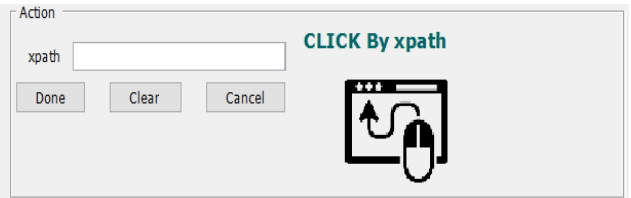


Figure 19 Click actions by XPATH

ii. ID

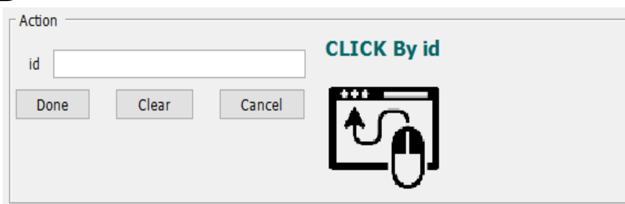


Figure 20 Click actions by ID

iii. Name:

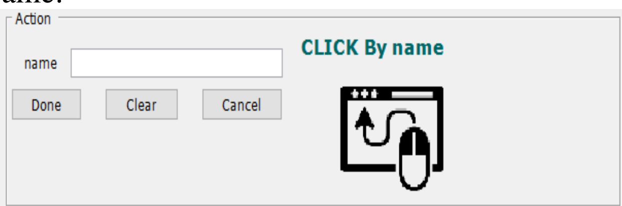


Figure 21 Click actions by Name

4. Add a set of conditions represented by radio buttons. Figure 22 below shows the radio buttons representing the condition code like >, <, Begin with. The testers can update or edit the condition that it's a DSTL notation, e.g., "CONTAINS," … etc.

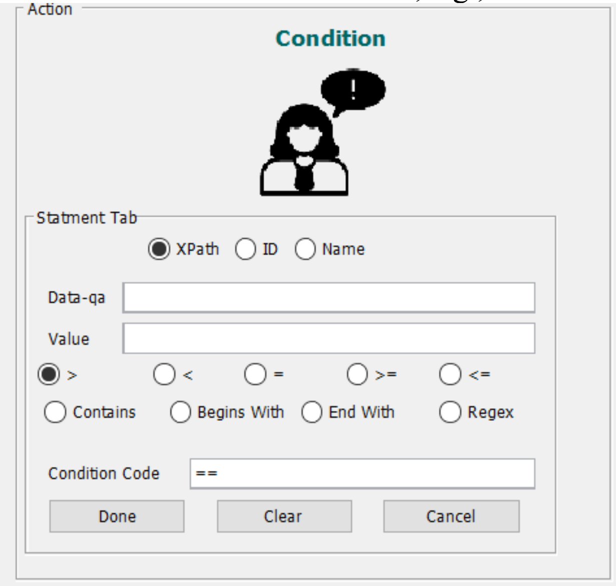


Figure 22 Condition figure

5. After the tester finished adding test steps by using steps 1-4 needs to select Generate Json File that will convert all steps to JSON format.
6. Click to generate selenium code that will create and generate test case contains all steps
7. For new test case need to repeat steps from 1-6

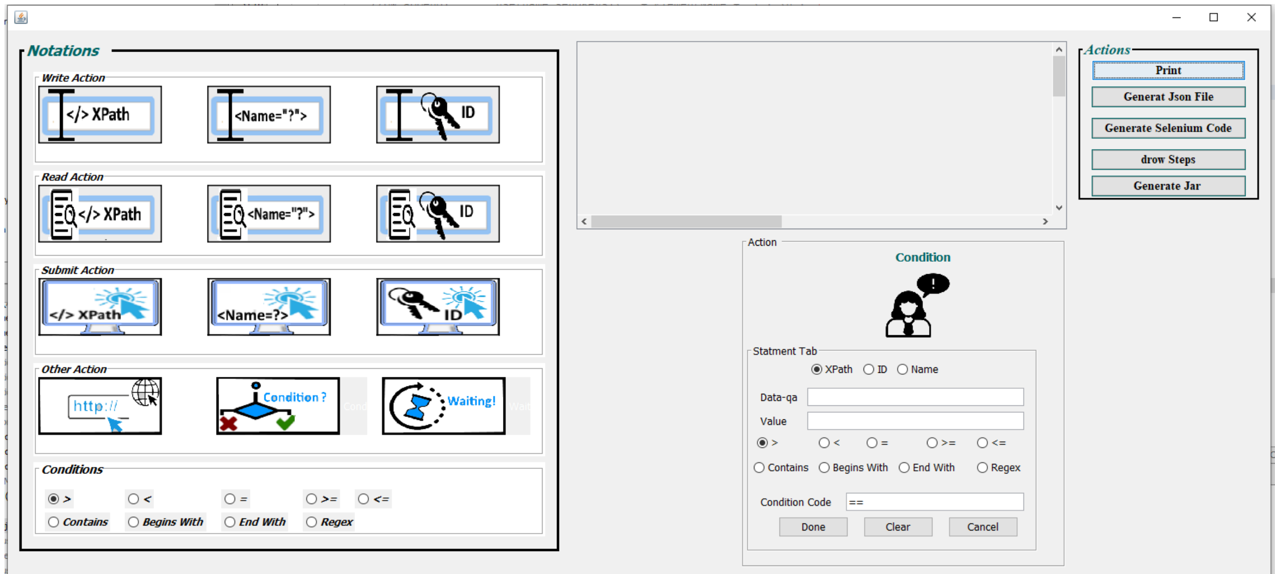


Figure 23 MAJD GUI

- **Generate JSON file for the steps:**

```
[
        {
            "step#:1":{
            "Element Name:":"URL ",
            "Step Action:":"URL",
            "Step number :":1,
            "XPATH Element:":"http:\/\/qaautomationdsvl.000webhostapp.com\/index.php",
            "Step discription:":"Step-1 _ Add URL to Test",

        }
        },
        {
            "step#:2":
            {
                    "Element Name:":"admin",
                    "Step Action:":"WRITE",
                    "Step number :":2,
                    "XPATH Element:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[1]",
                    "Step discription:":"Step-2 _ Add WRITE Action ",
                    "action Type:":"xpath"}
        },
        {
            "step#:3":
            {
                "Element Name:":"admin",
                "Step Action:":"WRITE",
                "Step number :":3,
                "XPATH Element:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[2]",
                "Step discription:":"Step-3 _ Add WRITE Action ",
                "action Type:":"xpath"
            }
        }
]
```

Figure 24 JSON steps file

- **Generated Selenium test case code**

```
Source  History

1
2
3   /**
4    *
5    * @author Generated By Automation Tool For QA
6    */
7
8   import java.util.concurrent.TimeUnit;
9   import org.openqa.selenium.By;
10  import org.openqa.selenium.WebDriver;
11
12  import org.openqa.selenium.WebElement;
13  import org.openqa.selenium.chrome.ChromeDriver;
14  import org.testng.Assert;
15
16  public class NewJFrame {
17      static WebDriver driver;
18      public static void main(String [] args) throws Exception {
19          System.setProperty("webdriver.chrome.driver", "C:\\Users\\admin\\Downloads\\chromedriver_win32 (1)\\chromedriver
20          driver=new ChromeDriver();
21          driver.manage().window().maximize();
22          //Step-1 _ Add URL to Test
23          step1();
24
25          //Step-2 _ Add WRITE Action
26          step2();
27
28          //Step-3 _ Add WRITE Action
29          step3();
30
31      }
32      private static void step1() throws Exception{
33
34          // **********##The step##-->  Step-1 _ Add URL to Test******
35         driver.get("http://qaautomationdsv1.000webhostapp.com/index.php");
36      }
37      private static void step2() throws Exception{
38          // **********##The step##-->  Step-2 _ Add WRITE Action ******
39         WebElement username=driver.findElement(By.xpath("/html/body/div[2]/div/form/div/div/input[1]"));
40         username.sendKeys("admin");
41      }
42      private static void step3() throws Exception{
43          // **********##The step##-->  Step-3 _ Add WRITE Action ******
44         WebElement username=driver.findElement(By.xpath("/html/body/div[2]/div/form/div/div/input[2]"));
45         username.sendKeys("admin");
46
47      }
48  }
```

Figure 25 Selenium code for test case

Chapter 5 Experimental design

The implemented approach has been evaluated using a case study, and this chapter will discuss the evaluation details, including participants' background, evaluation setups, evaluation procedure, and evaluation metrics.

5.1. Testers Background

The evaluation was conducted on a group of testers with different skills and different experience, 12 of them working on web projects and eight working on both web and mobile, related years of experience %50 of the testers have more than 11 years, 25 % between 6 and 10 and 25% less than six years. For the experience of automation, 25% have more than 11 years, and 20% less than two years. Java is the most automated language used, and 12 testers have already automated more than 11 test cases.

5.2. Evaluation Setup

The evaluation method was conducted on windows OS, 64-bit operating system, x64-based processor, Windows 10 Pro-version 21H1. With 16 G RAM, it was using NetBeans IDE. The case study was conducted using a tool to generate code for test cases called the MAJD tool. The tool is available as open-source on:

- **MAJD tool on Link git**[2]

The MAJD tool contains the main screen that has a lot of notation that is used to generate test cases. Participants will use the tool to generate a test case for their project by using DSVL and DSTL models.

- **Website**

I developed a website that contains many tabs about student registration to test everything related to the tool. Here you can find the link to website testing [3].

5.3. Environment setup

The testers did the following steps:

1- Open MAJD tool

---

[2] https://github.com/ref3t/MAJD-TOOL/tree/master link of the code
[3] http://qaautomationdsvl.000webhostapp.com/index.php website for testing

2- Add/Edit/delete test steps using notation
3- Generate JSON file that has all steps
4- Generate java (selenium) code for the steps
5- Execute the code and check the status.

5.4. Evaluation Metrics

- **Tester experience**

The testers' skills and level of experience were collected to determine the minimum skills needed for the testers to use the MAJD to generate test cases. The testers experience determines with the following list of factors:

  - Years of experience.
  - Experience background.
  - Years of experience in automation.
  - Experience in automation languages.
  - Number of projects.
  - Many test cases exist on the application of his project.
  - How many tests that he automates.

- **Ease of learning**

The case of learning metric was conducted and measured by observing the tester's mistakes and system failures while the tester is doing the task, also the time that he needs to discover how the tool works and use it.

The second part of the questions in Table 2 below focused on usability, failures, and user acceptance. These question tester will fill it after he finished the task, the following factor the determined Ease of use metric:

  - Problems while using the MAJD tool
  - Testers rating on the tool's usability level
  - Ability to understand how the tool works
  - Ability to understand the generated code
  - Participants rating complexity of using the tool to generate code
  - Participants rated the complexity of the generated code.

Chapter 6 Results and Discussion

This chapter presents the case study, results, and data discussion.

6.1. Participants experience

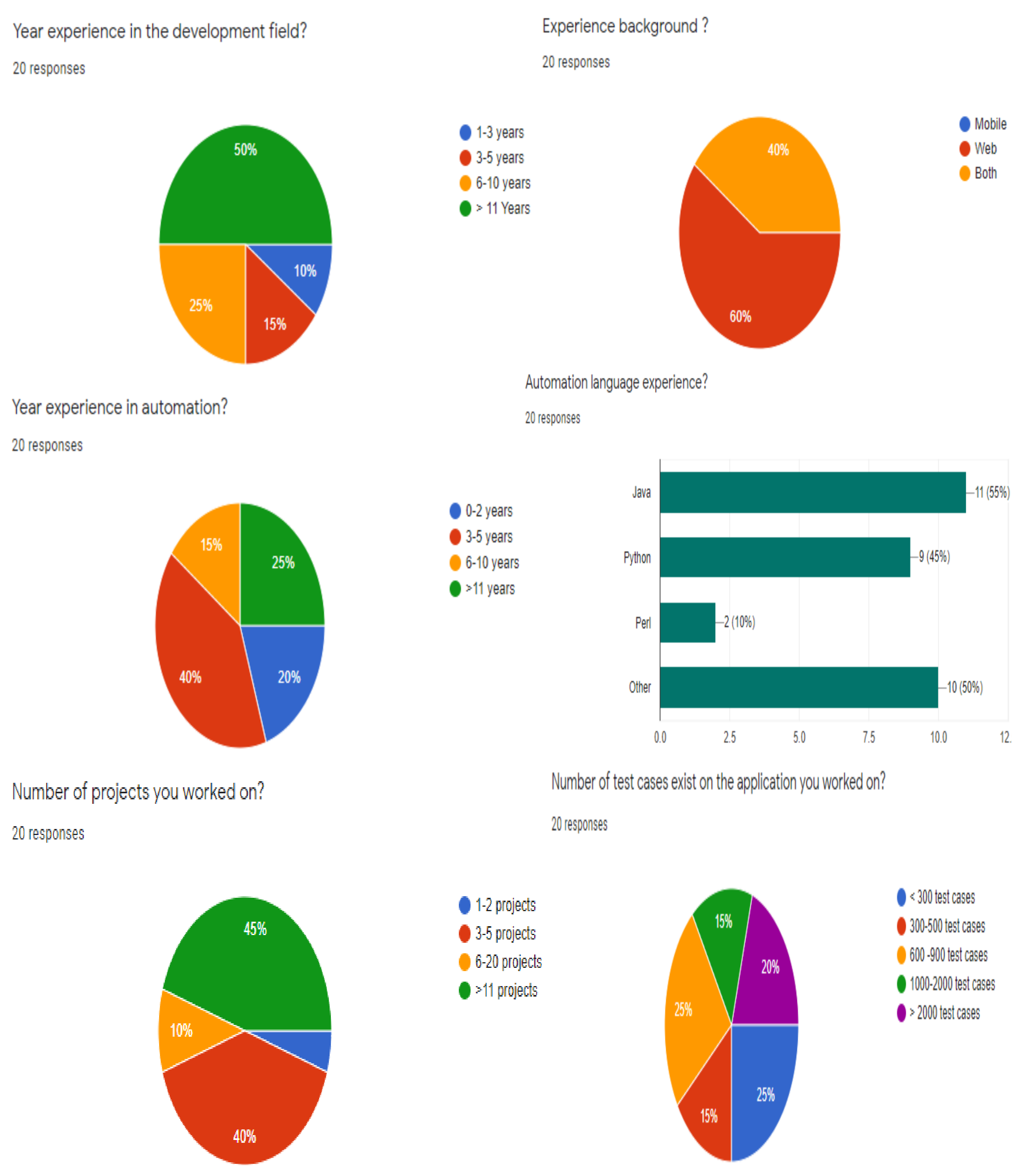
The participant's experience was collected using the first part of the questionnaire, and the table below shows the level of experience and background of the testers. The high level of background details for the testers was discussed in section 5.1 above.

The following Table 6-1 below shows participants' answers to a list of the questionnaire's first part questions.

Table 2 testers answers for tester's experience questions.

Year experience in the development field?	Experience background?	Year experience in automation?	Automation language experience?	Number of projects you worked on?	The number of test cases exists on the application you worked on?	How many automated tests have you built?
> 11 Years	Web	0-2 years	Other	>11 projects	300-500 test cases	6-10 tests
6-10 years	Both	6-10 years	Java;Python;Perl	3-5 projects	1000-2000 test cases	> 11 tests
3-5 years	Both	3-5 years	Java;Python	3-5 projects	1000-2000 test cases	> 11 tests
> 11 Years	Web	6-10 years	Java;Python	3-5 projects	> 2000 test cases	> 11 tests
6-10 years	Both	3-5 years	Python	3-5 projects	> 2000 test cases	> 11 tests
3-5 years	Web	3-5 years	Java;Python	3-5 projects	300-500 test cases	> 11 tests
1-3 years	Both	0-2 years	Java	1-2 projects	< 300 test cases	6-10 tests
> 11 Years	Web	0-2 years	Other	>11 projects	> 2000 test cases	6-10 tests
6-10 years	Web	6-10 years	Python;Perl;Other	3-5 projects	1000-2000 test cases	> 11 tests
1-3 years	Both	3-5 years	Java;Python	3-5 projects	600 -900 test cases	> 11 tests
> 11 Years	Web	3-5 years	Java	>11 projects	< 300 test cases	1-5 tests
> 11 Years	Web	>11 years	Other	>11 projects	600 -900 test cases	> 11 tests
6-10 years	Web	3-5 years	Java;Other	6-20 projects	< 300 test cases	> 11 tests
3-5 years	Both	3-5 years	Java; Other	3-5 projects	300-500 test cases	6-10 tests
> 11 Years	Web	0-2 years	Other	>11 projects	< 300 test cases	1-5 tests
> 11 Years	Web	>11 years	Other	>11 projects	< 300 test cases	> 11 tests
6-10 years	Both	3-5 years	Python	>11 projects	600 -900 test cases	6-10 tests
> 11 Years	Web	>11 years	Java;Other	>11 projects	> 2000 test cases	> 11 tests

> 11 Years	Web	>11 years	Python;Other	6-20 projects	600 -900 test cases	0 test
> 11 Years	Both	>11 years	Java	>11 projects	600 -900 test cases	> 11 tests
> 11 Years	Web	0-2 years	Other	>11 projects	300-500 test cases	6-10 tests
6-10 years	Both	6-10 years	Java;Python;Perl	3-5 projects	1000-2000 test cases	> 11 tests



How many automated test have you built?

20 responses

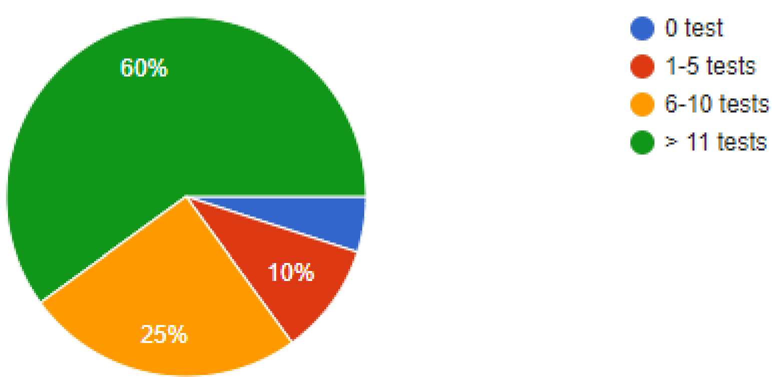


Figure 26 Participants' experience graph

All graphs in figure 26 show that the testers and participants have different experience levels, depending on the factor already listed in section 6.1. All testers could use the tool, add, edit, delete steps and generate code to the test, even the testers who don't have experience in automation.

The participants have different levels of experience in the automation field and different experience levels in web/mobile developments. The code generated by the tool is usable, Clear, simple, and understandable even for fresh testers or those from different backgrounds.

6.2. Ease of learning

This metric was measured by checking the participant's mistakes while doing tasks and the time needed to discover how the tool works and is used for generating test cases: users' mistakes, system failures, feedback, suggestions, and user acceptance questionnaire. Below table 2, table 3, and table 4 show the questionnaires part 2 results that already focused on the usability and the learnability of the approach and the testers' acceptance of the results.

Table 3 Participants answers questionnaire's part 2 questions

Participants/questions	did you face any problems while using the MAJD tool?	Easy to use the MAJD tool?	Do we have any problem understanding the tool especially the notations of what each icon represents?	Do we have any problem generating a test case using the MAJD tool?	High complexity to generate a test case using the MAJD tool?	Did you prefer to use the tool to generate the test case again?	You are satisfied with the quality of the test code generated by the tool?	You are satisfied with using MAJD Tool?
1	No	Agree	Strong Disagree	Yes	Disagree	Strongly agree	Agree	Strongly agree
2	No	Strongly agree	Strong Disagree	No	Strong Disagree	Strongly agree	Agree	Strongly agree
3	No	Strongly agree	Strong Disagree	No	Disagree	Strongly agree	Agree	Strongly agree
4	No	Strongly	Strong Disagree	No	Strong	Agree	Agree	Agree

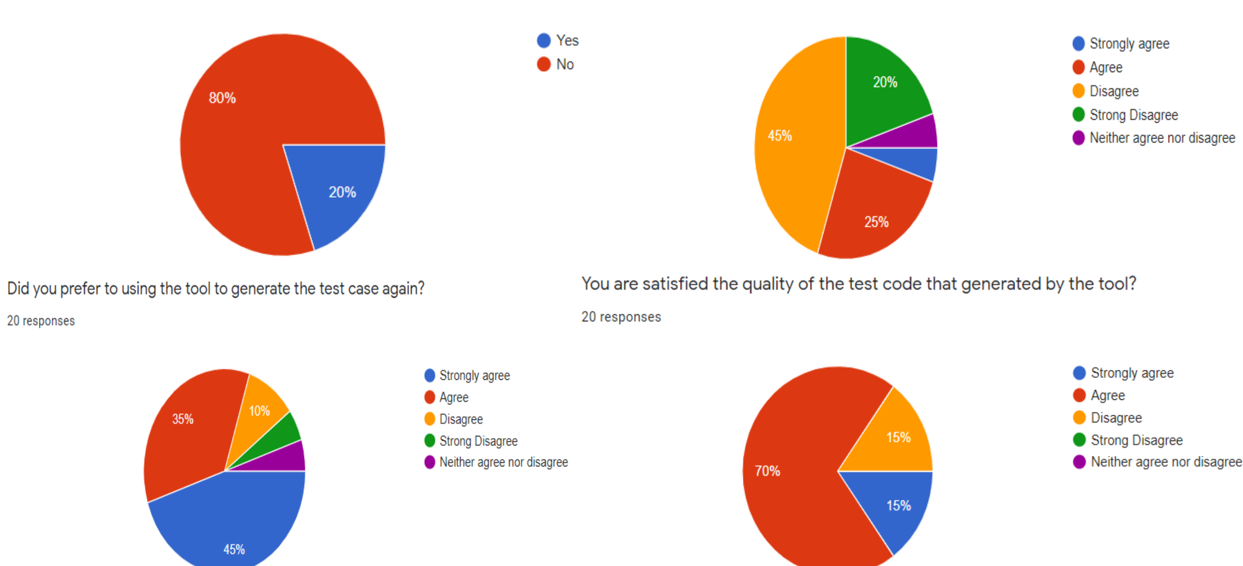
		agree			Disagree			
5	Yes	Strongly agree	Agree	No	Agree	Strongly agree	Agree	Strongly agree
6	No	Strongly agree	Disagree	No	Disagree	Strongly agree	Strongly agree	Strongly agree
7	Yes	Agree	Disagree	No	Disagree	Agree	Agree	Agree
8	No	Agree	Strong Disagree	No	Agree	Strongly agree	Strongly agree	Strongly agree
9	No	Agree	Disagree	No	Disagree	Agree	Agree	Agree
10	No	Agree	Disagree	No	Disagree	Agree	Disagree	Agree
11	No	Agree	Disagree	No	Strong Disagree	Strongly agree	Agree	Strongly agree
12	No	Agree	Disagree	No	Agree	Agree	Agree	Agree
13	No	Agree	Disagree	No	Strong Disagree	Strongly agree	Agree	Agree
14	No	Agree	Strongly agree	No	Agree	Agree	Agree	Agree
15	Yes	Agree	Agree	Yes	Neither agree nor disagree	Agree	Agree	Agree
16	No	Strongly agree	Strongly agree	No	Strongly agree	Strongly agree	Strongly agree	Strongly agree
17	No	Neither agree nor disagree	Agree	Yes	Agree	Disagree	Disagree	Disagree
18	No	Agree	Disagree	No	Disagree	Disagree	Agree	Agree
19	No	Agree	Disagree	No	Disagree	Neither agree nor disagree	Agree	Agree
20	No	Disagree	Agree	Yes	Disagree	Strong Disagree	Disagree	Strong Disagree

Table 4 Part 2 questions of the questionnaire after each of tester did his task. The score has been given from 1 to 5 representing Strongly Agree – Strongly Disagree.

# of question	**Question**	**Frequency %**				
		1	**2**	**3**	**4**	**5**
Q2	Easy to use the MAJD tool?	30%	60%	5%	0%	5%
Q3	Do we have any problem understanding the tool especially the notations what each icon represents?	10%	20%	45%	25%	0%
Q5	High complexity to generate a test case using MAJD tool?	5%	25%	45%	20%	5%
Q6	Did you prefer to using the tool to generate the test case again?	45%	35%	10%	5%	5%
Q7	You are satisfied the quality of the test code that generated by the tool?	15%	70%	15%	0%	0%
Q8	You are satisfied with using MAJD Tool?	40%	50%	5%	5%	0%

Table 5 Part 2 questions of the questionnaire.

# of question	**Question**	**Frequency %**	
		Yes	**No**
Q1	did you face any problems while using MAJD tool?	15%	85%
Q4	Do we have any problem generating a test case using MAJD tool?	20%	80%



You are satisfied with using MAJD Tool?
20 responses

Strongly agree
Agree
Disagree
Strong Disagree
Neither agree nor disagree

50%
40%

Figure 27 Ease of learning graphs

Table 3 and Table 4 present part 2 questions. It also presents the frequency of tester responses to each question. Q1 and Q2 Overall response was positive, agreeing that our approach is suitable for testers to generate test cases. Tester responses to Q4 indicate the tool generates test cases without any errors. 80% of the testers had positive feedback about problems appearing while using the approach.

Tester responses to Q6 provide tester satisfaction and prefer to use the tool for generating test cases many times. 80% of the testers had a positive reaction to the ease of learning of the approach (45 % Strongly Agree and 35% Agree). Q7 response was positive, agreeing with the quality of the code. 85% of participants responded with good feedback related to the quality of the code generated from the approach(15 % Strongly Agree and 85% Agree).

Tester's responses to Q8 indicate the overall acceptance of the approach. 90% of the testers had positive views on the tool's usefulness (40 % Strongly Agree and 50% Agree).

6.3. Usability Questions

The first three questions of the second part of the questionnaire target the user usability and learnability of the tool. Most of the testers (17/20) already confirmed that they did not face any problems while using the tool. Three participants mentioned that they faced problems while using the tool. Nineteen participants mentioned and confirmed that they did not have any problem understanding how the tool works and the generated code for the tests. Eighteen participants gave positive answers for the usability level questions, (6/20) marked it as strongly agree for easy to use, and (12/20) marked it as easy to use. The questionnaire results indicate the high usability and understandability of this tool. Figure 28 shows questions related to how the tool is easy to use.

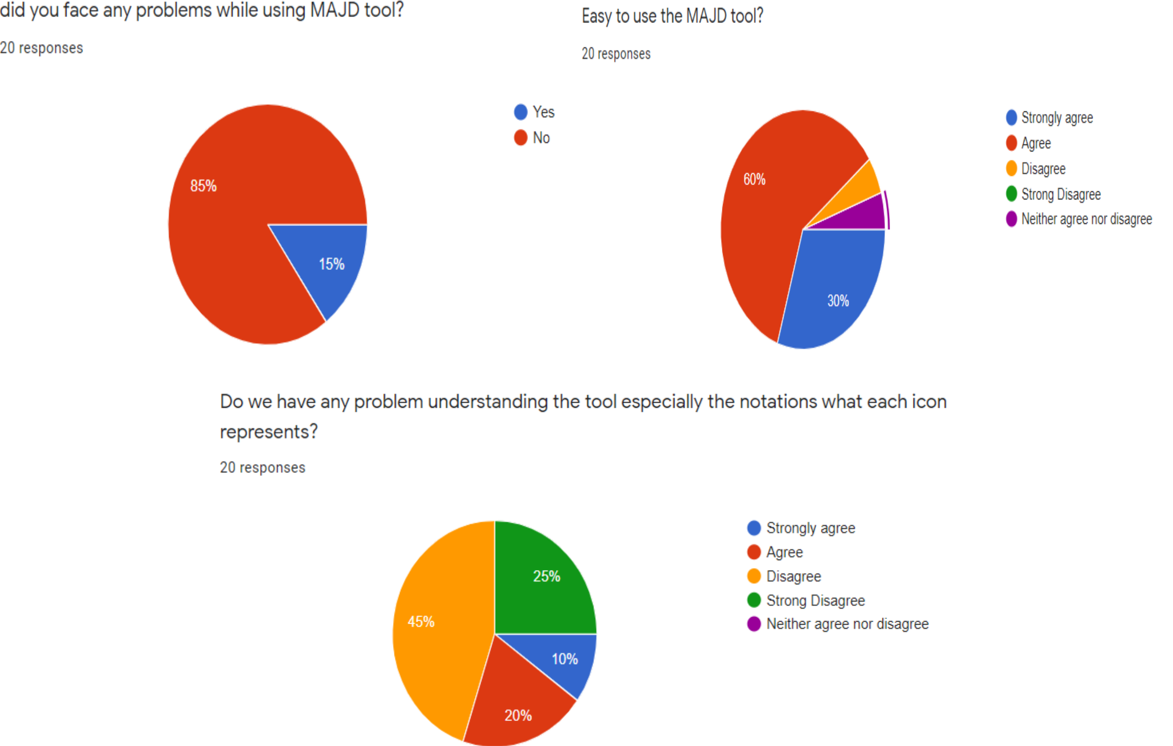


Figure 28 Usability questions

6.4. Failures, mistakes, and errors

To measure the usability of the tool as well as discover the mistakes and errors users might make while using MAJD, the following errors we already focused on

- System Errors or failures: which might have some system crashes, run time exceptions, or other errors that might be unexpected.
- User Mistakes: which could occur during using the tool, such as missing or misunderstanding of the functionality that caused to behave invalid generated test cases.
- Generated code errors: which could be syntax errors or invalid/corrupted generated code.

All participants did all required tasks using MAJD to generate test cases that all code of the already generated tests were ready to use. The already generated code is simple, easy to understand, and execute. Figure 27 and table 4 show the results about generated code and provide the effectiveness of MAJD on generating test cases.

Most of the participants could do the tasks without making a critical mistake. Even those who do not have automation experience could use the tool and understand the code. There are no system failures or errors in the generated code for the generated code, even syntax errors. The main issue that some participants already faced is that we have suggestions from the tool to detect all elements from the web page instead of the user adding the element manually. This will be moved to future work.

6.5. Models and notations evaluation

The MAJD takes the steps of the tests in JSON format that represents the steps data model, and it contains the steps and steps information. Manual inspection of all data of the steps includes all info for each test by using the JSON schema validator tool. Figure 29 and figure 30 show the write step model's results that contain all steps and information related to steps. The following steps that generated after adding all info from DSVL/DSTL and evaluated manually. All models and transformation models to models can find it in chapter 4.4, including input and output for each model.

- Write action (DSVL)
  - Steps Information(step model)

```
[
    {
        "step#:1":{
                "Element Name:":"admin",
                "Step Action:":"WRITE",
                "Step number :":1,
                "XPATH
Element:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[1]
",
                "Step discription:":"Step-1 _ Add WRITE Action
",
                "action Type:":"xpath"
            }
        ,
        "step#:2":{
                "Element Name:":"admin",
                "Step Action:":"WRITE",
                "Step number :":2,
                "XPATH Element:":"password",
                "Step discription:":"Step-2 _ Add WRITE Action
",
                "action Type:":"name"
            }
    }
]
```

  - Steps code(test case model)

```
private static void step1() throws Exception{
     // **********##The step##==>  Step-1 _ Add WRITE Action
******
       WebElement
username=driver.findElement(By.xpath("\/html\/body\/div[2]\/div\/
form\/div\/div\/input[1]\",")); 
       username.sendKeys("admin");
   }
   private static void step2() throws Exception{
       // **********##The step##==>  Step-2 _ Add WRITE Action
******
       WebElement
username=driver.findElement(By.name("password"));
       username.sendKeys("admin");
   }
```

- Read Action (DSVL)
  - Steps Information(step model)

```
[{
    "step#:3": {
        "Element Name:": "admin",
        "Step Action:": "READ",
        "Step number :": 3,
        "XPATH Element:":
"\/html\/body\/div[2]\/div\/form\/div\/div\/input[1]",
        "Step discription:": "Step-3 _ Add Read Action ",
        "action Type:": "xpath"
    }
}]
```

  - Steps code(test case model)

```
private static void step3() throws Exception{
       // **********##The step##==>  Step-3 _ Add Read Action
******

      WebElement
username=driver.findElement(By.xpath("\/html\/body\/div[2]\/di
v\/form\/div\/div\/input[1]"));

   }
```

- Submit action (DSVL)
  - Steps Information(step model)

```
[{
    "step#:1": {
        "Element Name:": "",
        "Step Action:": "CLICK",
        "Step number :": 4,
        "XPATH Element:": "login",
        "Step discription:": "Step-4 _ Add Click Action ",
        "action Type:": "name"
```

```
    }
}]
```

- Steps code(test case model)

```
private static void step4() throws Exception{
       // **********##The step##==>  Step-4 _ Add Click Action
******
      WebElement login=driver.findElement(By.name("login"));
      login.click();
   }
```

- Condition action (DSVL/DSTL)
  - Steps Information(step model)


```
[{
    "step#:1": {
        "Step Action:": "IF",
        "Step number :": 5,
        "Step discription:": "Step-5 _ Add IF Condition ",
        "data-qa:": "username",
        "action Type:": "name",
        "condition:": "=",
        "value:": "admin"
    }
}]
```


  - Steps code(test case model)

```
   private static void step5() throws Exception{
       // **********##The step##==>  Step-5 _ Add IF Condition
******
      WebElement elem=driver.findElement(By.name("username"));
      if
(!"admin".trim().equalsIgnoreCase(elem.getText().trim())) {
         throw (new Exception ( "exception value must be
equals"));
      }
   }
```

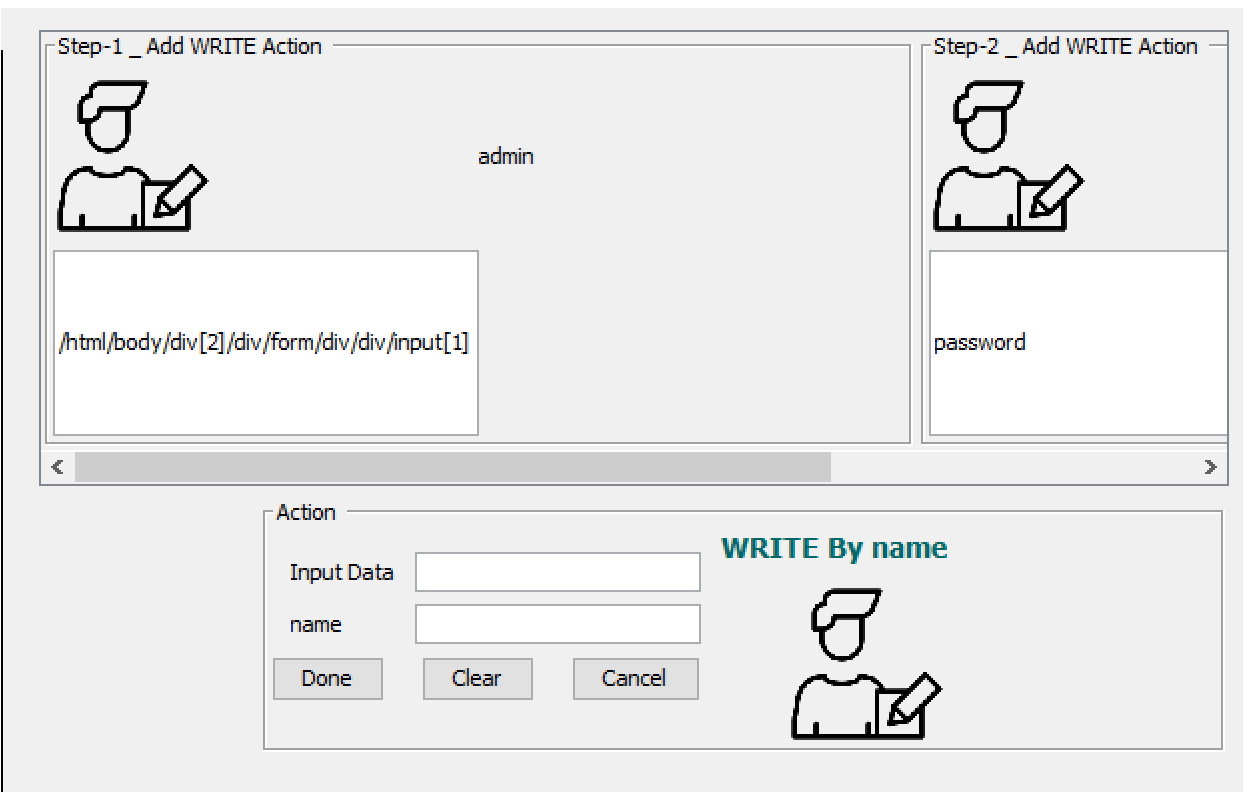


Figure 29 Steps Login

JSON JSONLint - The JSON Validator


```
[{
	"step#:1": {
		"Element Name:": "admin",
		"Step Action:": "WRITE",
		"Step number :": 1,
		"XPATH Element:": "\/html\/body\/div[2]\/div\/form\/div\/div\/input[1]",
		"Step discription:": "Step-1 _ Add WRITE Action ",
		"action Type:": "xpath"
	}
}, {
	"step#:2": {
		"Element Name:": "admin",
		"Step Action:": "WRITE",
		"Step number :": 2,
		"XPATH Element:": "password",
		"Step discription:": "Step-2 _ Add WRITE Action ",
		"action Type:": "name"
	}
}]
```


Validate JSON Clear

Results

valid JSON

Figure 30 Validation steps model

6.6. Threats to validity

The presented tool was evaluated using a case study and user evaluations questionnaire conducted on 20 participants. This case study provides many metrics about the participant experience needed to use, ease of learning, and the evaluation provides a user acceptance of the tool in helping the participants generate test cases instead of doing it manually.

Chapter 7 Conclusion and Future Work

7.1. Conclusion

In this research, we presented and provided a new fully automated code test case approach for the web application that aims to help testers generate test cases using model-based approach techniques. The approach employs model-based techniques that automatically generate test cases using Domain-Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL) to automatically generate test cases. Also, the model using the model to model transformation followed the model to model code generation.

A critical review of the background and related works presented. It concluded that there is no such solution available that helps testers automatically generate test cases, and already focused on this point.

A proof-of-concept tool was implemented and presented to measure the user acceptance, efficiency, and effectiveness of the approach used for generating code for the tests. In addition, the tool was evaluated using a case study and user evaluation study conducted on a group of 20 testers and developers from different experience levels. Participants used the MAJD tool to autogenerate selenium code for the tests of the web applications that were prepared for this study. They automatically generate tests using Domain-Specific Visual Language (DSVL) and Domain-Specific Textual Language (DSTL). All the results already discussed in chapter 6 that have many metrics include participants experience, ease of learning, user mistakes. And the testers demonstrated acceptance of the presented approach.

7.2. Future Work

In the future, we need to improve the implemented approach tool support by adding functionality to generate complicated tests and improving UI to provide higher usability and new features to the implemented tool. In addition, the author suggests adding a very important feature to select the web element automatically instead of doing it manually.

References:

[1] P. Kunte and D. Mane, "Automation Testing of Web based application with Selenium and HP UFT (QTP)", International Research Journal of Engineering and Technology (IRJET), vol. 0406-2017, no. 2395-0072, pp. 2579-2583, 2017. [Accessed 27 April 2019].

[2]Al-Zain, S., Eleyan, D., and Garfield, J. Automated User Interface Testing for Web Applications and TestComplete. In CUBE, ACM (2012), 350--354.

[3]Renu Patil, Rohini Temkar, “Intelligent Testing Tool: Selenium Web Driver”, irjet Volume 4, Issue 6, June 2017

[4]Manju Khari and Prabhat Kumar, "An extensive evaluation of search-based software testing: a review", Soft Computing (2017), 2017

[5] Sneha, K., & Malle, G. M. (2017). Research on software testing techniques and software automation testing tools. 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS), 77–81. IEEE.

[6] Vahid Garousi and Mika Mäntylä. "When and what to automate in software testing? A multi-vocal literature review". Vol. 76. Information and Software Technology, Apr. 2016. DOI: 10 . 1016 / j . infsof . 2016.04.015

[7] Milad Hanna, Amal Elsayed Aboutabl, Mostafa-Sami M. Mostafa,“Automated Software Testing Framework for Web Applications”, International Journal of Applied Engineering Research (IJAER), Vol. 13, no. 11, 2018, pp. 9758-9767

[8] P. Kunte and D. Mane, "Automation Testing of Web-based application with Selenium and HP UFT (QTP)", International Research Journal of Engineering and Technology (IRJET), vol. 0406-2017, no. 2395-0072, pp. 2579-2583, 2017. [Accessed 27 April 2019].

[9] Jawwad Ibrahim et al. "Emerging Trends in Software Testing Tools Methodologies: A Review". Vol. 17. International Journal of Computer Science and Information Security, Dec. 2019, pp. 108–112

[10] Kristian Wiklund et al. "Impediments for software test automation: A systematic literature review: Impediments for Software Test Automation". Vol. 27. Software Testing, Verification and Reliability, Sept. 2017, e1639. DOI: 10.1002/stvr.1639

[11] V. Garousi and F. Elberzhager, "Test automation: not just for test execution," IEEE Software, In Press, 2017.

[12] E. Vila, G. Novakova, and D. Todorova, "Automation testing framework for web applications with selenium webdriver: Opportunities and threats," in International Conference on Advances in Image Processing, ser. ICAIP 2017, 2017, pp. 144–150. [Online]. Available: http://doi.acm.org/10.1145/3133264.3133300

[13] Hanna, M., Aboutabl, A. E., & Mostafa, M. S. M. (2018). Automated software testing framework for web applications. International Journal of Applied Engineering Research, 13(11), 9758-9767. Retrieved from http://www.ripublication.com

[14] Nguyen H.P., Le H.A., Truong N.T. (2019) jFAT: An Automation Framework for Web Application Testing. In: Cong Vinh P., Alagar V. (eds) Context-Aware Systems and Applications, and Nature of Computation and Communication. ICCASA 2018, ICTCC 2018. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 266. Springer, Cham. https://doi.org/10.1007/978-3-030-06152-4_5

[15] Pinto, M., Gonçalves, M., Masci, P., & Campos, J. C. (2017). TOM: A Model-Based GUI Testing Framework. Lecture Notes in Computer Science, 155–161. doi:10.1007/978-3-319-68034-7_9

[16] Li, W.; Le Gall, F.; Spaseski, N. A Survey on Model-Based Testing Tools for Test Case Generation. In Tools and Methods of Program Analysis; Itsykson, V., Scedrov, A., Zakharov, V., Eds.; Springer International Publishing: Cham, Switzerland, 2018; Volume 779, pp. 77–89. doi:10.1007/978-3-319-71734-0_7. [CrossRef]

[17] Panthi V., and Mohapatra, D.P. 2017. An approach for dynamic web application testing using MBT.International Journal of System Assurance Engineering and Management. 2017; 8(2):1704-16.

[18] Ana C. R. Paiva, André Restivo, and Sérgio Almeida. Test case generation based on mutations over user execution traces. Softw. Qual. J., 28(3):1173–1186, 2020.

[19] Ferreira, S.M.A. (2019). Mutation-based web test case generation. Master's thesis.

[20] Gupta, N., Yadav, V., and Singh, M. (2018). Automated regression test case generation for web application: A survey. ACM Computing Surveys (CSUR), 51(4):87

[21] Barnett, S., Avazpour, I., Vasa, R., & Grundy, J. (2019). Supporting multi-view development for mobile applications. Journal of Computer Languages, 51, 88–96. doi:10.1016/j.cola.2019.02.001

[22] Radwan, A., & Zein, S. (2020). Model-Based Approach for Supporting Quick Caching at iOS Platform. International Journal, 9(4).

[23] Baek, Y.-M., & Bae, D.-H. (2016). Automated model-based Android GUI testing using multi-**level** GUI comparison criteria. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016. doi:10.1145/2970276.2970313

[24] M. Fischer, "Model-driven code generation for REST APIs," Modellgetriebene Code Generierung für REST APIs, 2015, doi: http://dx.doi.org/10.18419/opus-9803

Appendix A: Questionnaire

PART 1: Participants background questions:

Q1) Year experience in the development field?

a- 1-3 years b- 3-5 years c- 6-10 years d- >11 years

Q2) Experience background

a- Mobile b- Web c-Both

Q3) Year experience in automation?

a- 1-2 years b- 3-5 years c- 6-10 years d- >11 years

Q4) Automation language experience?

a- Java b- Python c- Perl d- others

Q5) Number of projects you worked on?

a- 1-2 projects b- 3-5 projects c- 6-10 projects d- >11 projects

Q6) Number of test cases exist on the application you worked on?

a- Less than 300 test cases b- 300-500 test cases c- 600 -900 test cases

d- 1000-2000 test cases f- >2000 test cases

Q7) How many automated test have you built?

a- 0 test b- 1-5 tests c- 6-10 tests d- >10 tests

The following Table 3 below shows a list of questionnaires' first part questions:

Table 6 - Part 1 interview question

Tester number	Year experience in the development field ➢ 1-3 years ➢ 3-5 years ➢ 5-10 years ➢ >10 years	Experience background ➢ Mobile ➢ Web ➢ Both	Year experience in automation ➢ 1-3 years ➢ 3-5 years ➢ 5-10 years ➢ >10 years	Automation language experience ➢ Java ➢ Python ➢ Perl	Number of projects you worked	Size of application you worked on ➢ Less than 300 test cases ➢ 300-600 test cases ➢ 600 -1000 test cases ➢ 1000-2000 test cases ➢ >2000 test cases
1						

2						

PART 2: Tool evaluation questions:

Q1) did you face any problems while using this tool?

a- Yes b- No

Q2) Easy to use the tool?

a- Strongly agree b- Agree c- Disagree

d- Strong Disagree e- Neither agree nor disagree

Q3) Do we have any problem understanding the tool especially the notations of what each icon represents?

a- Strongly agree b- Agree c- Disagree

d- Strong Disagree e- Neither agree nor disagree

Q4) Do we have any problem generating a test case using the tool?

b- Yes b- No

Q5) What is the complexity to generate a test case using the tool?

a- Strongly agree b- Agree c- Disagree

d- Strong Disagree e- Neither agree nor disagree

Q6) Did you prefer to use the tool to generate the test case again?

a- Strongly agree b- Agree c- Disagree

d- Strong Disagree e- Neither agree nor disagree

Q7) You are satisfied with the quality of the tests generated by the tool?

a- Strongly agree b- Agree c- Disagree

d- Strong Disagree e- Neither agree nor disagree

Q8) You are satisfied with using QA Automation Tool?

a- Strongly agree b- Agree c- Disagree

d- Strong Disagree e- Neither agree nor disagree

The following Table 4 below shows a list of questionnaires' second part questions.

Table 7 - Part 2 approach evaluation

Tester number	did you face any problem while using this approach ? ➔ Yes ➔ No	Easy to use the approach ? ➢ Strongly agree ➢ Agree ➢ Disagree ➢ Strong Disagree ➢ Neither agree nor disagree	high complexity to generate a test case using the approach ? ➢ Strongly agree ➢ Agree ➢ Disagree ➢ Strong Disagree ➢ Neither agree nor disagree	Do we have any problem generating test cases using the approach ? ➢ Strongly agree ➢ Agree ➢ Disagree ➢ Strong Disagree ➢ Neither agree nor disagree	What is the complexity to generate a test case using the approach ? ➢ Strongly agree ➢ Agree ➢ Disagree ➢ Strong Disagree ➢ Neither agree nor disagree	) You are satisfied with the quality of the tests generated by the tool?	You are satisfied with using QA Automation Tool?
1							
2							

Appendix B: Generated code for a sample project using MAJD

Example 1:

- **JSON file code for tests**


```
[{
   "step#:1": {
      "Element Name:": "URL ",
      "Step Action:": "URL",
      "Step number :": 1,
      "XPATH Element:": "http:\/\/localhost\/SMTA\/index.php",
      "Step discription:": "Step-1 _ Add URL to Test",
      "action Type:": ""
   }
}, {
   "step#:2": {
      "Element Name:": "admin",
      "Step Action:": "WRITE",
      "Step number :": 2,
      "XPATH Element:": "username",
      "Step discription:": "Step-2 _ Add WRITE Action ",
      "action Type:": "name"
   }
}, {
   "step#:3": {
      "Element Name:": "admin",
      "Step Action:": "WRITE",
      "Step number :": 3,
      "XPATH Element:": "\/html\/body\/div[2]\/div\/form\/div\/div\/input[2]",
      "Step discription:": "Step-3 _ Add WRITE Action ",
      "action Type:": "xpath"
   }
}, {
   "step#:4": {
      "Element Name:": "admin",
      "Step Action:": "CLICK",
      "Step number :": 4,
      "XPATH Element:": "submit",
      "Step discription:": "Step-4 _ Add Click Action ",
      "action Type:": "name"
   }
}, {
```

```
    "step#:5": {
      "Element Name:": "admin",
      "Step Action:": "CLICK",
      "Step number :": 5,
      "XPATH Element:": "\/\/*[@id=\"menu-top\"]\/li[4]\/a",
      "Step discription:": "Step-5 _ Add Click Action ",
      "action Type:": "xpath"
    }
  }, {
    "step#:6": {
      "Step Action:": "IF",
      "Step number :": 6,
      "Step discription:": "Step-6 _ Add IF Condition ",
      "data-qa:": "\/html\/body\/div[2]\/div[1]\/div[1]\/div\/h1",
      "action Type:": "xpath",
      "condition:": "=",
      "value:": "Student Registration"
    }
  }, {
    "step#:7": {
      "Element Name:": "admin",
      "Step Action:": "CLICK",
      "Step number :": 7,
      "XPATH Element:": "\/html\/body\/div[2]\/div[2]\/div\/div[2]\/div\/div\/div[2]\/div\/table\/tbody\/tr[1]\/td[4]\/a[2]\/button",
      "Step discription:": "Step-7 _ Add Click Action ",
      "action Type:": "xpath"
    }
  }, {
    "step#:8": {
      "Step Action:": "IF",
      "Step number :": 8,
      "Step discription:": "Step-8 _ Add IF Condition ",
      "data-qa:": "\/html\/body\/div[2]\/div[2]\/div\/div[2]\/font",
      "action Type:": "xpath",
      "condition:": "=",
      "value:": "Student record deleted ffffff!!"
    }
  }]
```


- **Generated Code for a test**

```
/**
 *
```

```
 * @author Generated By Automation Tool For QA
 */

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.Assert;


public class NewJFrame {
  static WebDriver driver;
  public static void main(String [] args) throws Exception {
     System.setProperty("webdriver.chrome.driver",
"C:\\Users\\admin\\Downloads\\chromedriver_win32 (1)\\chromedriver.exe");
     driver=new ChromeDriver();
     driver.manage().window().maximize();
     //Step-1 _ Add URL to Test
     step1();

     //Step-2 _ Add WRITE Action
     step2();

     //Step-3 _ Add WRITE Action
     step3();

     //Step-4 _ Add Click Action
     step4();

     //Step-5 _ Add Click Action
     step5();

     //Step-6 _ Add IF Condition
     step6();

     //Step-7 _ Add Click Action
     step7();

     //Step-8 _ Add IF Condition
     step8();

  }
```

```
private static void step1() throws Exception{

    // **********##The step##==>  Step-1 _ Add URL to Test******

   driver.get("http://localhost/SMTA/index.php");

 }

private static void step2() throws Exception{

    // **********##The step##==>  Step-2 _ Add WRITE Action ******

   WebElement username=driver.findElement(By.name("username"));
   username.sendKeys("admin");

 }

private static void step3() throws Exception{

    // **********##The step##==>  Step-3 _ Add WRITE Action ******

   WebElement
username=driver.findElement(By.xpath("/html/body/div[2]/div/form/div/div/input[2]"));
   username.sendKeys("admin");

 }

private static void step4() throws Exception{

    // **********##The step##==>  Step-4 _ Add Click Action ******

   WebElement login=driver.findElement(By.name("submit"));
   login.click();

 }

private static void step5() throws Exception{
```

```java
    // **********##The step##==>  Step-5 _ Add Click Action ******

    WebElement login=driver.findElement(By.xpath("//*[@id=\"menu-top\"]/li[4]/a"));
    login.click();

  }

  private static void step6() throws Exception{



    // **********##The step##==>  Step-6 _ Add IF Condition ******

    WebElement elem=driver.findElement(By.xpath("/html/body/div[2]/div[1]/div[1]/div/h1"));
    if (!"Student Registration".trim().equalsIgnoreCase(elem.getText().trim())) {
      throw (new Exception ( "exception value must be equals"));
    }

  }

  private static void step7() throws Exception{



    // **********##The step##==>  Step-7 _ Add Click Action ******

    WebElement
login=driver.findElement(By.xpath("/html/body/div[2]/div[2]/div/div[2]/div/div/div[2]/div/table/
tbody/tr[1]/td[4]/a[2]/button"));
    login.click();

  }

  private static void step8() throws Exception{



    // **********##The step##==>  Step-8 _ Add IF Condition ******

    WebElement
elem=driver.findElement(By.xpath("/html/body/div[2]/div[2]/div/div[2]/font"));
    if (!"Student record deleted ffffff!!".trim().equalsIgnoreCase(elem.getText().trim())) {
      throw (new Exception ( "exception value must be equals"));
```

```
      }

   }
}
```

Example 2

- **Generate JSON file for the steps:**

```
[
    {
      "step#:1":{
      "Element Name:":"URL ",
      "Step Action:":"URL",
      "Step number :":1,
      "XPATH Element:":"http:\/\/qaautomationdsvl.000webhostapp.com\/index.php",
      "Step discription:":"Step-1 _ Add URL to Test",

    }
    },
    {
      "step#:2":
      {
            "Element Name:":"admin",
            "Step Action:":"WRITE",
            "Step number :":2,
            "XPATH Element:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[1]",
            "Step discription:":"Step-2 _ Add WRITE Action ",
            "action Type:":"xpath"}
    },
    {
      "step#:3":
      {
         "Element Name:":"admin",
         "Step Action:":"WRITE",
         "Step number :":3,
         "XPATH Element:":"\/html\/body\/div[2]\/div\/form\/div\/div\/input[2]",
         "Step discription:":"Step-3 _ Add WRITE Action ",
         "action Type:":"xpath"
      }
    }
]
```

- **Generated Selenium test case code**

```
/**
 *
 * @author Generated By Automation Tool For QA
 */

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.Assert;


public class NewJFrame {
  static WebDriver driver;
  public static void main(String [] args) throws Exception {
      System.setProperty("webdriver.chrome.driver",
"C:\\Users\\admin\\Downloads\\chromedriver_win32 (1)\\chromedriver.exe");
      driver=new ChromeDriver();
      driver.manage().window().maximize();
      //Step-1 _ Add URL to Test
      step1();

      //Step-2 _ Add WRITE Action
      step2();

      //Step-3 _ Add WRITE Action
      step3();

  }

  private static void step1() throws Exception{



      // *********##The step##==>  Step-1 _ Add URL to Test******
```

```
    driver.get("http://qaautomationdsvl.000webhostapp.com/index.php");

  }

  private static void step2() throws Exception{



     // **********##The step##==>  Step-2 _ Add WRITE Action ******

    WebElement
username=driver.findElement(By.xpath("/html/body/div[2]/div/form/div/div/input[1]"));
    username.sendKeys("admin");

  }

  private static void step3() throws Exception{



     // **********##The step##==>  Step-3 _ Add WRITE Action ******

    WebElement
username=driver.findElement(By.xpath("/html/body/div[2]/div/form/div/div/input[2]"));
    username.sendKeys("admin");

  }}
```