VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Minh Nguyen

# DEVELOPING AUTOMATED UI TESTING

Technology and Communication
2023

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information and Technology

## ABSTRACT

| | |
|---|---|
| Author | Minh Nguyen |
| Title | Developing Automated UI testing |
| Year | 2023 |
| Language | English |
| Pages | 28 + 4 Appendices |
| Name of Supervisor | Smail Menani |

The primary objective of the thesis was to investigate and evaluate different methodologies and strategies for automating the testing of user interfaces (UI) in software systems.

The research methodology used was design science research. The thesis process is divided into two main phases: Literature review and Development. A literature review was conducted to investigate the best practices and appropriate techniques for automating user interface testing. In the empirical case study, an automated system for user interface tests was developed as part of the new Wärtsilä application development. The UI testing was then integrated into automating regression testing. The feedback and suggestion during this phase were incorporated into the final product.

Along with test development, test reports can be generated. It is essential that test reports are precise and provide sufficient information for evaluating the error's cause.

As a result of the study, a new UI testing code was developed and implemented on the actual software product at Wärtsilä. The test provided partly consistent and reliable results with various scenarios and cases. The regression test was done repeatedly to avoid unexpected errors during development.

Keywords: User interface testing, test automation, and testing

## ACKNOWLEDGEMENTS

As an embedded software developer with minimal web development and testing tools knowledge, I felt very proud of completing this thesis before the project started. My coding style and expertise have also significantly developed while working on the thesis, thanks to my teammates' reviews. The knowledge acquired through this study will serve as a strong foundation for any future endeavors in automation.

I want to express my special gratitude to my beloved team Walkie Talkie at Wärtsilä, who helped me along this journey by providing the tools, valuable documents, and code reviews. Special thanks to Markus, Staffan, Tommi, Antti, Jarkko, Tulika, Thomas, Tobias, and Arek for your invaluable support. To all my colleagues at Wärtsilä, thank you for your direct or indirect support towards my thesis.

Special thanks to my thesis supervisor Smail Menani for the priceless advice, direction, and support. With your steering, I would gain while writing my thesis. Thank you to all teachers in the Information Technology program at VAMK for nourishing my knowledge during the study.

My deepest thanks go to my family. Thank you for always being by my side.

Vaasa, April 10, 2023

Minh Nguyen

**CONTENTS**

## TABLE OF FIGURES

**LIST OF APPENDICES**

## ABBREVIATIONS

| | |
|---|---|
| A&C | Automation and Control |
| C&R | Capture & Replay |
| CD | Continuous Delivery |
| CI | Continuous Integration |
| DOM | Document Object Model |
| E2E | End to End |
| GUI | Graphical User Interface |
| HTTP | Hypertext Transfer Protocol |
| PI | Planning Increment |
| UI | User Interface |

# 1 INTRODUCTION

Regression testing often entails performing predefined test cases on each new version to confirm that the product satisfies its requirements. [1] However, as an extensive system in Wärtsilä, it seems impossible to execute regression tests manually before each release. Therefore, by automating repetitive tasks, test automation may be beneficial for regression testing.

An impediment of the thesis is that automation UI testing has never been used in any project at A&C development. Consultation from the UX/UI team gave unpromising results. Automation UI testing for web applications was required due to the importance of long-term benefits for the product after release and reduced time and resources required for manual testing, allowing developers or manual testers to focus on other tasks.

The thesis aims to build an automated UI testing system for the new web application to improve the defect-finding ability. Test execution should be convenient with clear test reports. On the author's part, the thesis helps widen his knowledge about automation testing framework and fosters the author's personal and professional growth in this area. Time management, evaluating processes, and developing coding logic are also obviously achieved after the study.

Firstly, conducted methodologies and their information are described. While conducting empirical research is explored in more depth, a literature review is only touched upon briefly. The literature review covers the definition of automation testing with different testing classifications. The development explains in detail the solution disclosed in the literature review in practice and proves its reasoning. Finally, the conclusion gives a general view of the findings and presents possibilities for future work.

# 2 METHODOLOGY

The chapter gives an overview of the thesis's implementation. The selected research approach and the reason it was chosen are explained in detail, along with the project design.

The systematic review was actively used throughout the study. Google Scholar and Theseus.fi were primarily used to look for appropriate papers. To gain more insight into the problem, articles, and references at the end of searched research were also read to find further details.

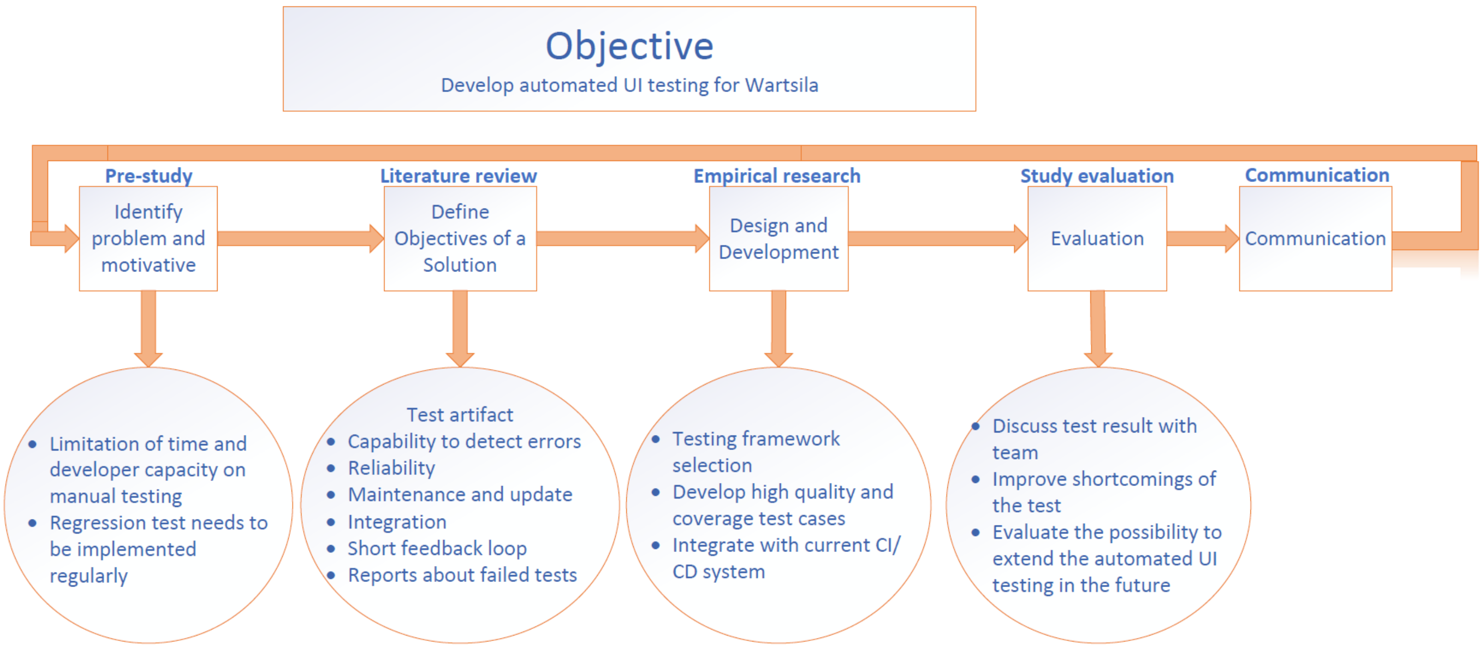
## 2.1 Research Approach

The proper research method for developing automation UI testing would depend on several factors, including the specific goals of the testing, the resources available, and the time constraints. After reviewing carefully, the author settled on design science research as the most suitable.

According to Ken Peffers and his colleagues [2], the design science research process is divided into five parts: Identify the Problem & Motivate, Define the Objectives of a Solution, Design and Development, Evaluation, and Communication. An overview of the process and each part will be discussed next.

## 2.2 Research Design

The research design of this study is split into several parts mentioned above. Each part of the design process will be discussed in Figure 1.



Objective
Develop automated UI testing for Wartsila
Pre-study
Identify problem and motivative
Literature review
Define Objectives of a Solution
Empirical research
Design and Development
Study evaluation
Evaluation
Communication
Communication
Limitation of time and developer capacity on manual testing
Regression test needs to be implemented regularly
Test artifact
Capability to detect errors
Reliability
Maintenance and update
Integration
Short feedback loop
Reports about failed tests
Testing framework selection
Develop high quality and coverage test cases
Integrate with current CI/CD system
Discuss test result with team
Improve shortcomings of the test
Evaluate the possibility to extend the automated UI testing in the future

**Figure 1.** The design science research process came from the DSRM Process Model of Ken Peffers (2)

### 2.2.1 Identifying Problems and Motivation

Developers ‘capacities reserved for maintenance are quite limited. Then the time for testing and addressing problems must be minimized as least as possible. In addition, when the application is handed over to the customer, it is highly inconvenient for developers to notice and solve the problem without an automated regression test. Therefore, it was decided that regression testing should be automated so that manual testing resources may be dedicated to more exploratory testing.

### 2.2.2 Define the Objectives of a Solution

The goal of the automation is to free up manual testing time and resources for use on exploratory testing, maintenance, new features, and currently under-tested areas. The overarching goal is broken down into smaller, more specific goals that would be easier to track. They are explained and discussed as follows.

- Capability to detect errors: Capability to detect errors is a good indicator of how practical the tests are in preventing defects from being deployed in the product system. [3] Even if tests fail to detect bugs, they are still helpful since they assure developers that their code will perform as expected.
- Reliability: All tests should behave in the same way, failing if anything goes wrong and passing if the tested functionality behaves as anticipated. Flaky tests can sometimes be avoided, but the testing system should behave as accurately as possible.
- Maintenance and update: Even though software evolved continually with new features, the test should be easy to maintain and update with little effort.
- Integration: The test should be part of the extensive system and be able to integrate with CI/CD.
- Short feedback loop: Feedback from the test (both positive and negative) should be processed as fast as possible after changes by developers
- Report about the failed test: Clear report should indicate which one is successful, which one failed, and why it failed. Test reports can be included in the final report sent to the customer.

### 2.2.3 Design and Development

Our web application was written mainly with the TypeScript programming language. Several programming options were added in the developing process. Some proprietary tools, such as TestCafe Studio, Ranorex Studio, or TestComplete, are robust and support cross-platform. However, their license prices are relatively minor compared to the cost of working hours spent on manual testing or developing the test automation system. Since the author has been involved in the project development from the early days and understands its structure and code, Cypress – an open-source end-to-end testing tool designed for modern web test automation was chosen as the primary tool for the study. It provides a quick, dependable, and user-friendly method for testing web applications, making it easier to identify and resolve errors before they affect end users. The Development part will mention a deeper explanation of how to create the test with Cypress. The last part of Development will describe CI integration and test reports.

### 2.2.4 Evaluation

The evaluation was conducted as part of the development process. Almost immediately after the completion of the initial tests, automated user interface testing was implemented, and these tests were continuously evaluated and enhanced. The developer frequently examined and analyzed codes for cleanliness, maintainability, and scalability during the development.

### 2.2.5 Communication

During the timeframe of this research, the development of automation UI testing for the application has been finished. Feedback from team members, supervisors, and manager has been gathered and incorporated into the system. Automation UI testing has become an integral part of the testing process for new application development.

# 3 LITERATURE REVIEW

Automated testing has been proposed as one solution to the problems with manual regression testing since automated tests can run faster and more often, decreasing the need for test case selection, and thereby raising quality while reducing manual effort [4]. However, because of its higher upfront cost in building the testing infrastructure and limitation of scope, automated user interface testing cannot replace human efforts and become a silver bullet in the industry. This literature review will compare and study different testing methods to determine the best approach and execute our case study's automated user interface test most effectively.

The categories of testing methods for automated UI testing are given in Figure 2 below based on Leotta [5]. A further explanation of each method will be discussed later, with examples in subsequent sections.

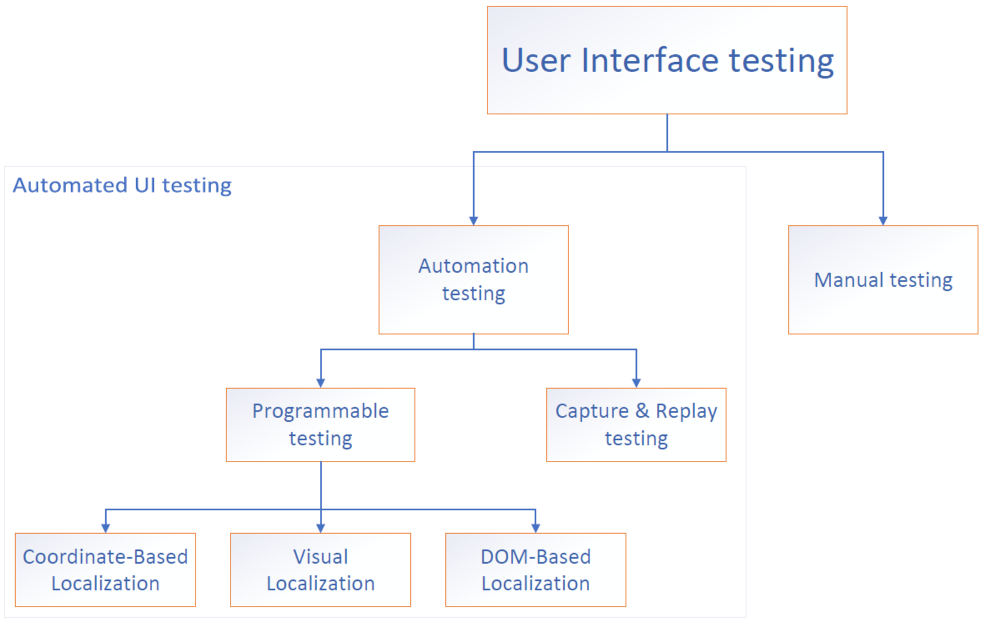


**Figure 2.** The categories based on Leotta [5] and Leivo [3]

In the next part, capture and reply testing will be discussed. Then Programmable Web Testing and Visual Localization are analyzed deeply, followed by the end-to-end testing description used in the case study. Lastly, the pros and cons of the different testing methods are mentioned in the last part.

## 3.1 Capture & Replay Testing

Capture & Replay (C&R) test cases are accessible and need no specialized testing expertise. C&R tools have been created to verify the accuracy of interactive applications (GUI or Web applications). The tool records the user's actions on explored Web pages, such as key presses and mouse clicks, in a script, enabling a session to be replayed automatically without further user input. Lastly, a test case is generated by including one or more assertions in the recorded script. By repeating a specified script on a modified Web Application Under Test (WAUT), capture/replay technologies provide automated regression testing [6]. Example of C&R Testing is shown in Appendix 1.

Nevertheless, according to Leotta and his colleagues' research, programmable tests are initially more expensive than capture and replay tests due to automated code development depending on user inputs. Unfortunately, a slight change in the GUI might disrupt a previously recorded test case, and the inability to reuse code in capture and replay tests leads to more extraordinary maintenance expenses, making it challenging to adopt this technique in our case study. Furthermore, if a programmable approach is adopted, the process of test suite evolution will become more straightforward and require less effort after at least two software releases, according to Leotta's conclusion. [6]

## 3.2 Programmable Web Testing

Contrary to Capture & Replay, Programmable Web testing is based on the manual development of a test script. Leotta [6] pointed out that UI elements such as buttons, input fields, or links can be automatically tested in different cases: Coordinate-Based Localization, DOM-Based Localization, and Visual Localization. Coordinate-Based Localization method is extremely obsolete because this method uses a screen recorder to compare the web page element during test case replays, which require precise application layout and screen resolution. If the layout or resolution changes, the UI element coordinates may also change, causing the tests to fail. Therefore, only two other methods are discussed, which are more robust and reliable for identifying UI elements.

### 3.2.1 Visual Localization

Visual localization is a method that locates and interacts with user interface elements by analyzing visual appearance. This method includes obtaining and analyzing screenshots of the UI components and identifying them using image recognition techniques. Visual localization is more flexible than DOM-Based Localization in some cases when UI elements are dynamic and challenging to obtain. In addition, because of using an image recognition algorithm, Visual localization can provide high accuracy in locating UI elements and is more reliable than traditional Coordinate-Based Localization. [6]

However, visual localization also has some limitations. The test can be sensitive to visual elements such as color, font size, or position, which are supposedly extraneous to the test case. Test performance using visual localization can be slower than other techniques since it needs two phases to execute: image recognition training and testing. Hence, spending significant capacity and resources on this method could be more practical.

### 3.2.2 DOM-Based Localization

In web applications and other structures, user interface components are found by their IDs or information accessible in the Document Object Model (DOM). During development, these elements are specified in the source code and accessible to the testing tool. This technique is called DOM-Based Localization. DOM-Based Localization identifies and locates UI components using their characteristics and properties, such as their id, class, name, and tag name. These properties may generate selectors and XPath expressions uniquely identifying the component of the page. [7]

In our case study, DOM-Based Localization is the most suitable approach due to its robust, flexible, and cost-effective technique for locating and interacting with user interface elements on web pages in the long run. Besides, the author participated in the development of the project from the early days and deeply understood the source code of the project. Cypress framework was used, introduced as the example above, to build and execute automated user interface tests. The details of the testing process will be mentioned further in the Development part.

## 3.3 End-to-end Testing

Test cases will be executed sequentially during testing; earlier tests may influence subsequent tests. Thus, end-to-end (E2E) testing is the optimal testing design for our system.

End-to-end testing is a kind of black box testing that verifies, from the user's perspective, that the application performs as intended from beginning to finish. It entails testing the complete application process from beginning to end to verify that all components function as intended and that the application fulfills the requirements and user expectations. [8]. E2E testing is advantageous since it can conduct tests quicker than anyone, and its findings can be automatically reproduced. In addition, E2E testing may be conducted unsupervised, saving significant testing time and costs. One or more test cases can be derived from a single test scenario by specifying the data for each step (for example, username=John.Doe) and the expected results (i.e., defining the assertions). The execution of each test case can be automated by implementing a test script following existing approaches (for example, Programmable and DOM-based localization) [7].

Broadly speaking, E2E testing can be approached by two classifications: test script implementation and web page elements localization [7]. There are several tools that support E2E testing, such as Selenium, Sikuli, or Cypress. As discussed in Chapter 3.2.2 about DOM-Base Localization, Cypress is chosen as an end-to-end framework for our web application.

## 3.4 Summary

In the previous chapters, a variety of testing methods were covered. In this section, a conclusion of all automated testing techniques is summarized and compared in further detail about their advantages and disadvantages. Manual testing is also considered, even though it was not mentioned above. The results of the comparison are summarized in Table 1. Those found in the literature review will be applied and evaluated in the Development part.

Automated user interface testing begins with the decision of what to automate. As pointed out in [4], not all user interface tests can be automated; most of them fail with a new version of the software under testing. Therefore, they suggest using automated testing for regression testing primarily. Manual testing still plays an important role and should be noticed, especially in exploratory testing. As a result, a combination of both manual and automated testing can help ensure that an application is thoroughly tested and performs as expected. [9]

**Table 1:** Pros and Cons of different user interface testing methods

| **Method** | **Pros** | **Cons** |
|---|---|---|
| **Manual testing** | Low initial cost; high degree of adaptability; good for exploratory testing | Time-consuming; vulnerable to human error; hard to run regressively |
| **Capture & Replay Testing** | Low initial cost; fast and efficient way to create test scripts; | Fragile; unreliability for a consistent run; high maintenance required |

| | | |
|---|---|---|
| | no programming skill required | |
| **Programmable Web Testing** | Robust; low maintenance cost; integrable with other automated testing tools | High initial cost; programming skill required |

Capture & Replay Testing is the easiest way to start automating user interface testing since no programming skill is required, and effortless to make a script. However, as mentioned above, since a script is generated through capture and replay testing, any changes to the UI or application may require manual updates to the script. This can be time-consuming and may make the script less reliable. Thus, over time, capture and replay scripts can become challenging to maintain, leading to increased maintenance costs for the project. If the program does not change often, this method can still be applied in some basic automated tests.

Programmable testing is divided into three sub-methods: Coordinate-Based Localization, DOM-Based Localization, and Visual Localization. As mentioned in Chapter 3.2, Coordinate-Based Localization is so fragile and complex to implement with different screen resolutions. According to research by Leotta [5], DOM-Based Localization takes less time to develop, less flaky test, and has low maintenance cost compared to visual localization. There are a few different hypotheses that might explain these findings. The IDs or classes of some elements are hard coded in the-source code but may have different styles based on responsive display or status, such as buttons and the value displayed on the screen; these elements can easily access and verify using the DOM-Based Localization testing method. In a visual approach, these various styles may make the test broken.

Moreover, visual localization uses an image recognition algorithm, which may be inaccurate when several same-looking elements are on the page. In addition, some test cases require different actions, which makes the visual approach hard to implement. Although there are advantages such as those mentioned above, there are also disadvantages to using DOM-Based Localization. These test suits cannot be written by non-technical testers, knowledge of source code is required. The flaky test can still happen if the underlying structure of the web page changes. Finally, each approach has its strengths and weaknesses. Visual Localization fits with the tester, who needs to become more familiar with the source code and vice versa.

Test coverage and test execution are also important factors that need to be considered when building an automated UI testing system. It is vital to ensure that the testing system can adequately cover all critical aspects of the application to identify and address any potential issues. As Berner highlighted, it is necessary to do the tests often and keep them in excellent condition since it is more costly to repair them later. [10]. Test prioritization also needs analyzing when building test suits. Some test cases are independent, while others are not. Then, an optimized UI testing sequence should be implemented to save time and make it easier to obtain feedback. Moreover, the necessary tests should be run to get developers' rapid feedback.

# 4 DEVELOPMENT

This chapter describes how an automated user interface testing system was built and integrated into our web application testing. The development part is divided into three subchapters. The first subchapter presents an overview of web application development. Then the following subchapter is the main focus of the study, the development of automated user interface testing. Lastly, test evaluation and the outcome of the test development are mentioned in the last subchapter.

## 4.1 Overview of Web Application Development

After several iterations in close collaboration with the A&C team members, some crucial functionalities of the application were ready for operation and testing. Meetings and multiple discussions were held through application development. The purpose of meetings was to review the process, gain feedback and improvement ideas, and steer the tool development project when needed.

## 4.2 Development of an Automated User Interface Testing System

As agreed with the General Manager and System Architect, the automated user interface testing for web applications was chosen for development in this thesis. Due to confidential policies, the names of test suits and tools were made up and hidden.

Figure 3 represents the architecture of the test automation system. It is also applied in other application development of the A&C department. When a developer commits a new change in the version control system, a new build based on the change is labeled as either successful or failed, depending on linters or unit tests running during the build step. If it is successful, the developer is allowed to commit the change in the staging area to the local repository. If a new pull request is needed, the CI system will build the code in a virtual machine based on the latest successful build. If it is successful, a developer can merge the change into the target branch (approval from other reviews must also be agreed upon clearly).

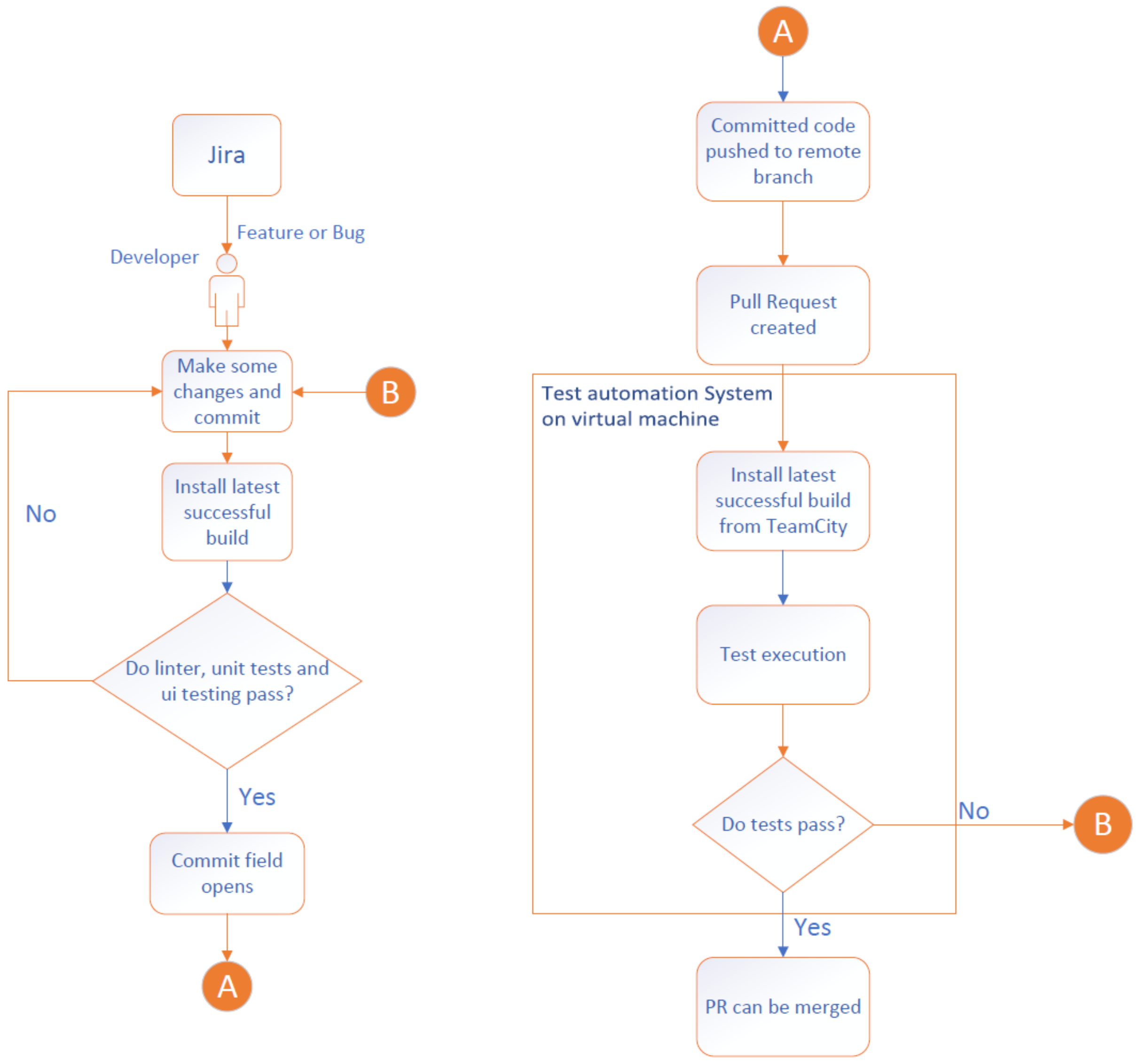

Jira
Feature or Bug
Developer
Make some changes and commit
B
No
Install latest successful build
Do linter, unit tests and ui testing pass?
Yes
Commit field opens
A
A
Committed code pushed to remote branch
Pull Request created
Test automation System on virtual machine
Install latest successful build from TeamCity
Test execution
Do tests pass?
No
B
Yes
PR can be merged

**Figure 3.** The architecture of the test automation system.

Various types of linter and unit testing are applied in our project to ensure its flawless execution. However, this chapter presents the description of how the automated user interface tests using Cypress were built and integrated into our CI system to make them run regressively.

### 4.2.1 Developing Cypress Automated User Interface Testing

Cypress is executed in the same run loop as the application. There is a Node server process underneath Cypress. Cypress and the Node process interact, synchronize, and continuously conduct tasks on each other's behalf. Access to both portions (front and back) allows developers to react to the application events in real-time while doing actions needing higher permission outside the browser. Cypress also functions at the network layer by dynamically reading and modifying web traffic. This allows Cypress to alter all incoming and outgoing browser traffic and any code that may impede its ability to automate the browser. [11]

Because of the large number of manual tests to be conducted in this PI, to limit the scope of the thesis, only one test suit will be executed, which is believed to be performed on any software package. Other test suits can be built based on this sample test case.

**Activation test**

The test objective is to verify that activate action functions as intended and application is in the specific state when Activate action is triggered. Activation test suit involves testing in numerous situations. Thus, the status of desired functionalities and some UI elements must be continuously checked throughout the testing process. Due to the limited scope of the thesis and multiple scenarios, a small test case is depicted in Figure 4 below.

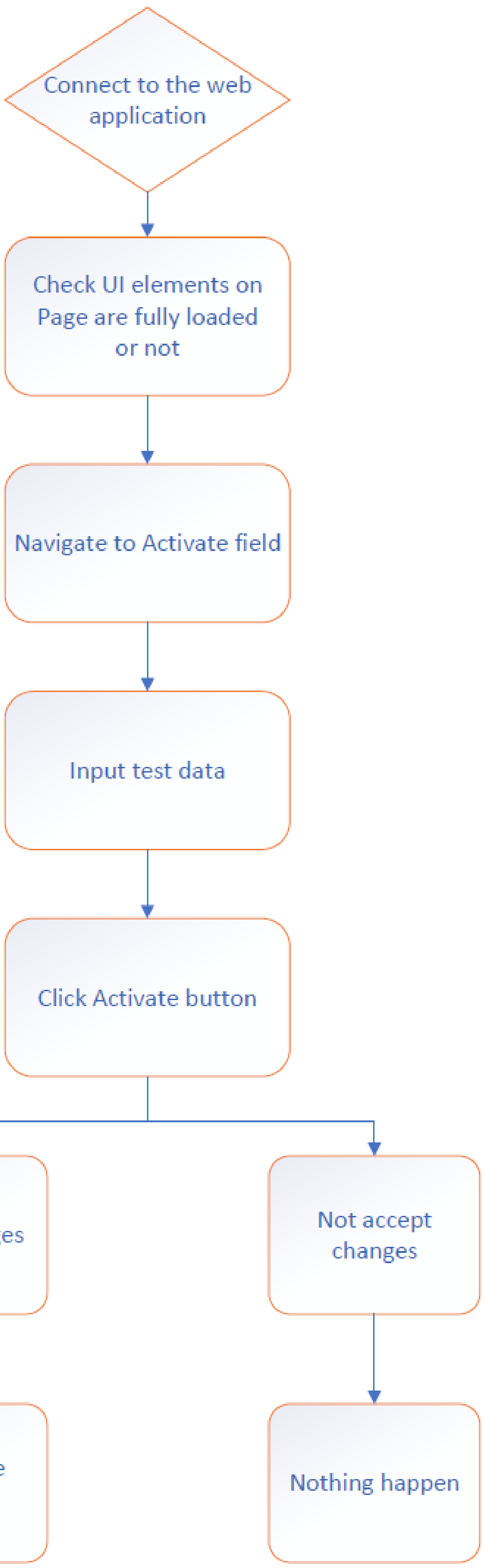
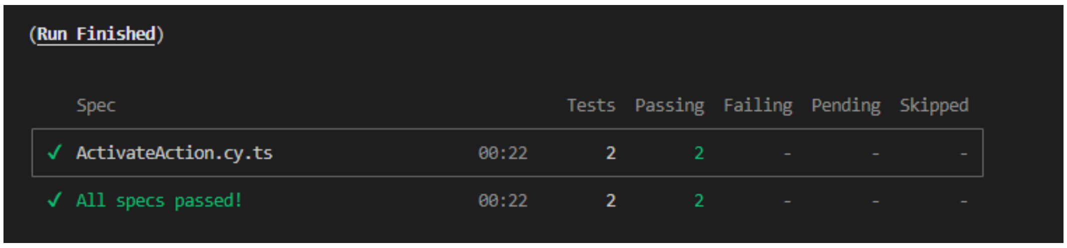

Connect to the web application
Check UI elements on Page are fully loaded or not
Navigate to Activate field
Input test data
Click Activate button
Accept changes
Not accept changes
Activate the action
Nothing happen

**Figure 4.** Testing process of Activate action

The content of web application is extracted from an XML file; therefore, it must be processed and loads the UI components gradually. We must determine whether the web application is connected. If not, the test fails due to a connection problem. After the application is connected and all UI components have loaded, the activation function will be tested. When the user clicks on the action that needs to activate, a dialog will pop up that requires the user to change and accepts user input. The user will be prompted to confirm their changes before proceeding. The application state will not change when the Cancel button is pressed; it will revert to the condition it was in when the page was assumed. Confirm will initiate the activation process. When a certain amount of time has passed after the activation sequence has completed its run, the UI components and some statuses will be examined to confirm that the application is in a proper state. The test specifications can be obtained from Appendix 4.

## Test Reports



```
(Run Finished)

      Spec                                        Tests  Passing  Failing  Pending  Skipped
  ┌──────────────────────────────────────────────────────────────────────────────────────────┐
  │ ✓  ActivateAction.cy.ts               00:22        2        2        -        -        - │
  └──────────────────────────────────────────────────────────────────────────────────────────┘
    ✓  All specs passed!                  00:22        2        2        -        -        -
```

**Figure 5.** Test reports

The test took about 22 seconds to finish all scenarios in the given example. Since only 1 test suit was performed, the amount of time is not much less than performing a manual test, which took about two minutes to run. However, applying it to the whole system may save a tremendous amount of time and effort. Cypress also supports video generators, which help improve the testing process by providing a visual representation of the testing process, making it easier to identify and fix bugs, and can be used as documentation after the product is finalized.

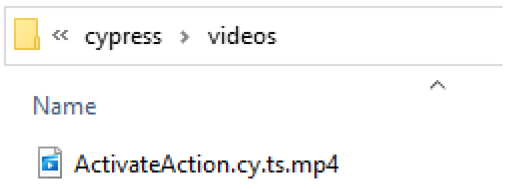
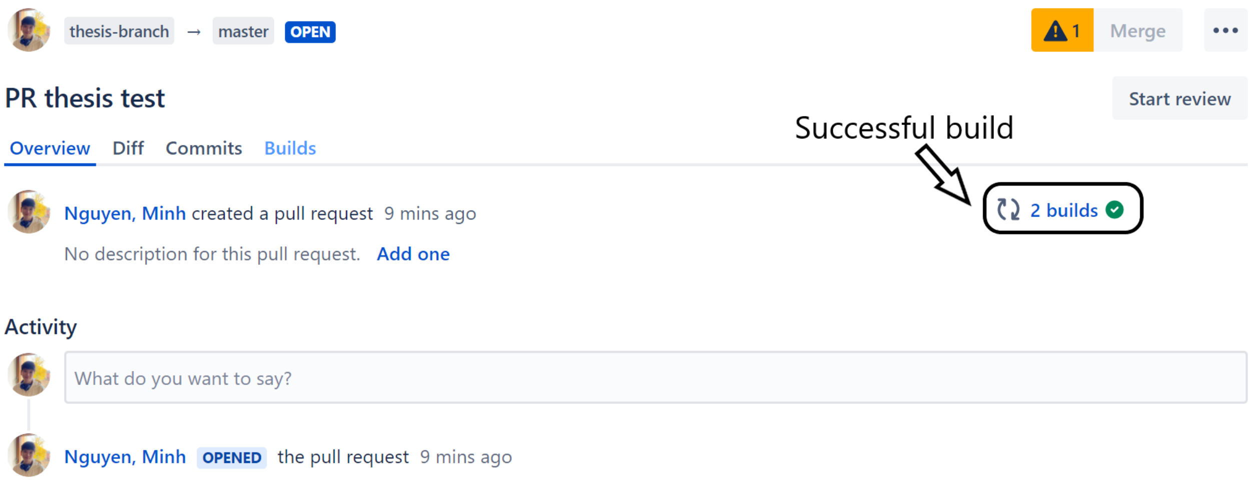


**Figure 6.** Videos of processed test

### 4.2.2 Integrating with CI/CD

This thesis uses a continuous integration server to build and test the automated UI testing system. Because of the scope of the thesis, the instruction and details on how to set up docker or CI system configuration will not be mentioned.

CI system will be run when a new pull request is created and for nightly regression tests. As illustrated below, regression testing needs much work to build up the rig test and CI configuration, then pull request verification.



PR thesis test

Nguyen, Minh created a pull request 9 mins ago

No description for this pull request. Add one

What do you want to say?

Nguyen, Minh OPENED the pull request 9 mins ago

**Figure 7.** Pull request with a successful build

Although the automated UI testing system has been integrated into the CI system, it has yet to be implemented due to missing the running server on the CI system, which is exceptionally crucial for the operation of the application.

# 5 DISCUSSION AND CONCLUSION

This is the last chapter of the research, and its objective is to discuss and summarize the thesis. Different testing methods were compared and applied to an existing testing system in the Development part. The thesis was presented to team members, stakeholders, and the system architect at a demo meeting. Everyone was pleased with the thesis and recommended some features that may be improved and replaced in the future.

In this chapter, the automated UI testing system will be evaluated based on the described requirements in Chapter 2.2.2. In addition, the existing system still has several shortcomings; however, due to the scope of this thesis, some suggestions will be provided for the future growth of the case study system.

## 5.1 Test Evaluation

The results of the testing system have partly satisfied the goals specified in Chapter 2.2.2. The automated UI testing system ran through all test cases and scenarios in only a few minutes, interacted with the server side to control and execute the functionality of web application, which manually endured a few hours to complete.

The automated UI testing system was capable of detecting errors in the user interface. Both successful and failed messages were shown when executing test suits. Since testing UI interacts directly with HTML, it is straightforward to comprehend source codes and maintain and create future test suites. Based on the sample test case described in the Development chapter, several test suites have been constructed. Lastly, Cypress provided excellent support for test reporting, with built-in reporting capabilities and support for third-party tools (screenshots, video recordings of the test runs).

However, so far, the reliability of the system must also be validated. Multiple test runs have been undertaken; however, the findings needed to be more consistent, and flaky tests still occurred. Various factors, such as the browser version, cache, or page load time, might cause this. This will make it easier and more convenient to examine pull requests. Furthermore, the short feedback loop factor can be built but has yet to be applied to the system since the source code has not been optimized and the test was quite fragile. The integration of automated UI testing into the CI/CD pipeline has yet to be accomplished in the Development area due to objective reasons related to the desired tested application. Recommendations for those drawbacks will be presented in the following subchapter.

## 5.2 Future Development

Before delivering the automated UI testing tool to the case organization's production, a substantial amount of work still needs to be done.

Multiple test runs occurred to verify the reliability of the test, but the result was not promising. Only 23/35 attempts were passed ultimately; other times, some test suits failed during the process. One of the most frequent errors was lost connection with server during test. Currently, the mechanism of web application when lost connection happens is wait for a short amount of time then automatically reconnect to the server and reset the state. However, this mechanism makes the test extremely obsolete since the test uses states of the application to determine a proper HTML to compare, especially when web application loses connection during middle of the test. In addition, the HTML elements were not hardcoded in the application source code. Then sometime HTML elements 'names were generated differently to the desired ones by undefined reason. Thus, further improvements are necessary to make the test more robust and trustworthy.

Moreover, some test suits require testing multiple screens simultaneously. In actual production, not only one application is used but also multiple ones are used to execute functionality on web application. Multiple tabs testing is under development from Cypress team. Therefore, developers must find workarounds to overcome this problem. Some internal features of some applications in A&C

departments also create obstacles for automated UI testing, which needs to be handled by exploratory testing.

The automated UI testing system was not completely integrated to CI/CD to perform nightly test. Since the application requires a running system to handle all functionalities, a feature to create and connect the web application to the running system on a virtualization server or an independent simulation in the desired application should be implemented to fulfill the CI integration.

## REFERENCES

[1] S. M. Mohammad, "Automation Testing in Information Technology," *International Journal of Creative Research Thoughts (IJCRT),* 2015.

[2] T. T. ,. M. A. R. &. S. C. Ken Peffers, "A Design Science Research Methodology for," *Journal of Management Information Systems,* pp. 45-77, 2007.

[3] T. Leivo, "Automating user interface testing: Case study at Finnish Transport Agency," 2017.

[4] Emil Borjesson and Robert Feldt, "Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation,* 2012.

[5] Leotta, M., Clerissi, D., Ricca, F., & Tonella, P., "Visual vs. DOM-Based Web Locators: An Empirical Study," *Web Engineering,* pp. 322-340, 2014.

[6] D. C. F. R. P. T. Maurizio Leotta, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution.," pp. 272-281, 2013.

[7] M. C. D. R. F. &. T. P. Leotta, "Approaches and Tools for Automated End-to-End Web Testing," *Advanced in Computers,* pp. 193-237, 2016.

[8] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè,, "Automated gui refactoring and test script repair," *Proceedings of the First International Workshop on end-to-end Test Script Engineering,* pp. 38-41, 2011.

[9] Emil Al´egroth, Robert Feldt, and Lisa Ryrholm, "Visual gui testing in practice: challenges, problemsand limitations. Empirical Software Engineering, 20," 2015.

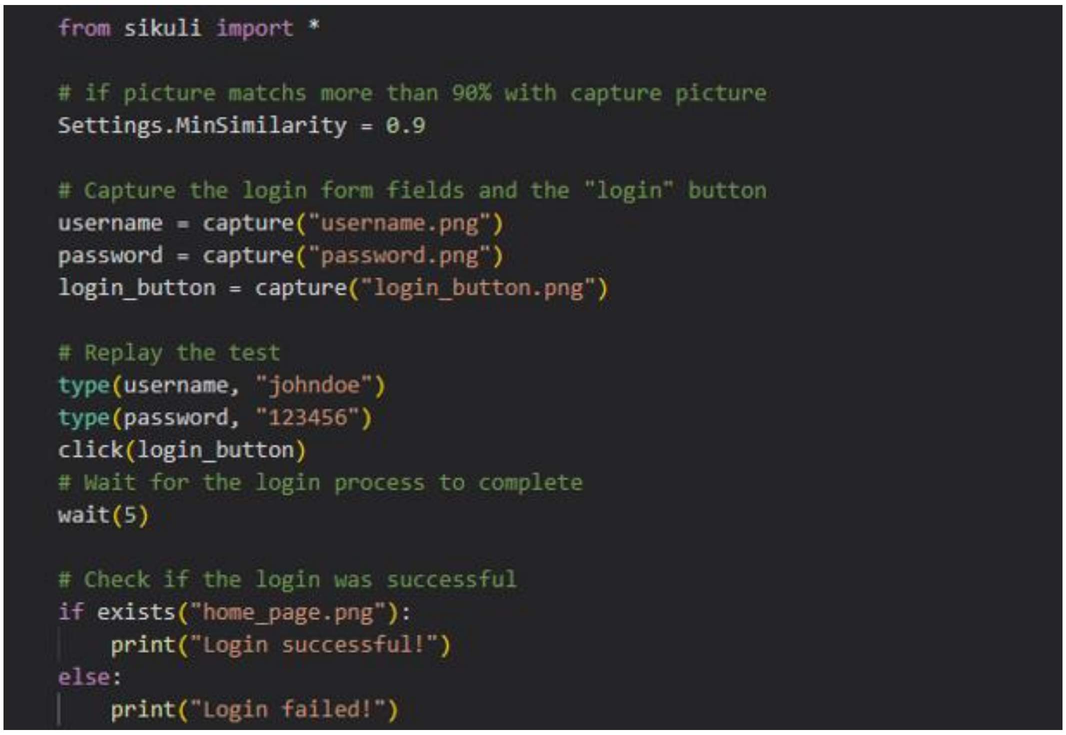
[10] Stefan Berner, Roland Weber, and Rudolf K Keller, "Observations and lessons learned from automated testing," *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference,* pp. 571-579, 2005.

[11] Cypress, "Cypress," [Online]. Available: https://docs.cypress.io/guides/overview/key-differences.

[12] S. K. G. N. D. &. G. P. Gupta, "DOM-based Content Extraction of HTML Documents," *Proceedings of the Twelfth International Conference on World Wide Web* - *WWW '03.,* 2003.

## APPENDIX 1: Example of UI testing using Capture & Replay Testing

The C&R testing method is commonly used with the Sikuli library. A typical example of how C&R test case implementation is shown in Figure 5.

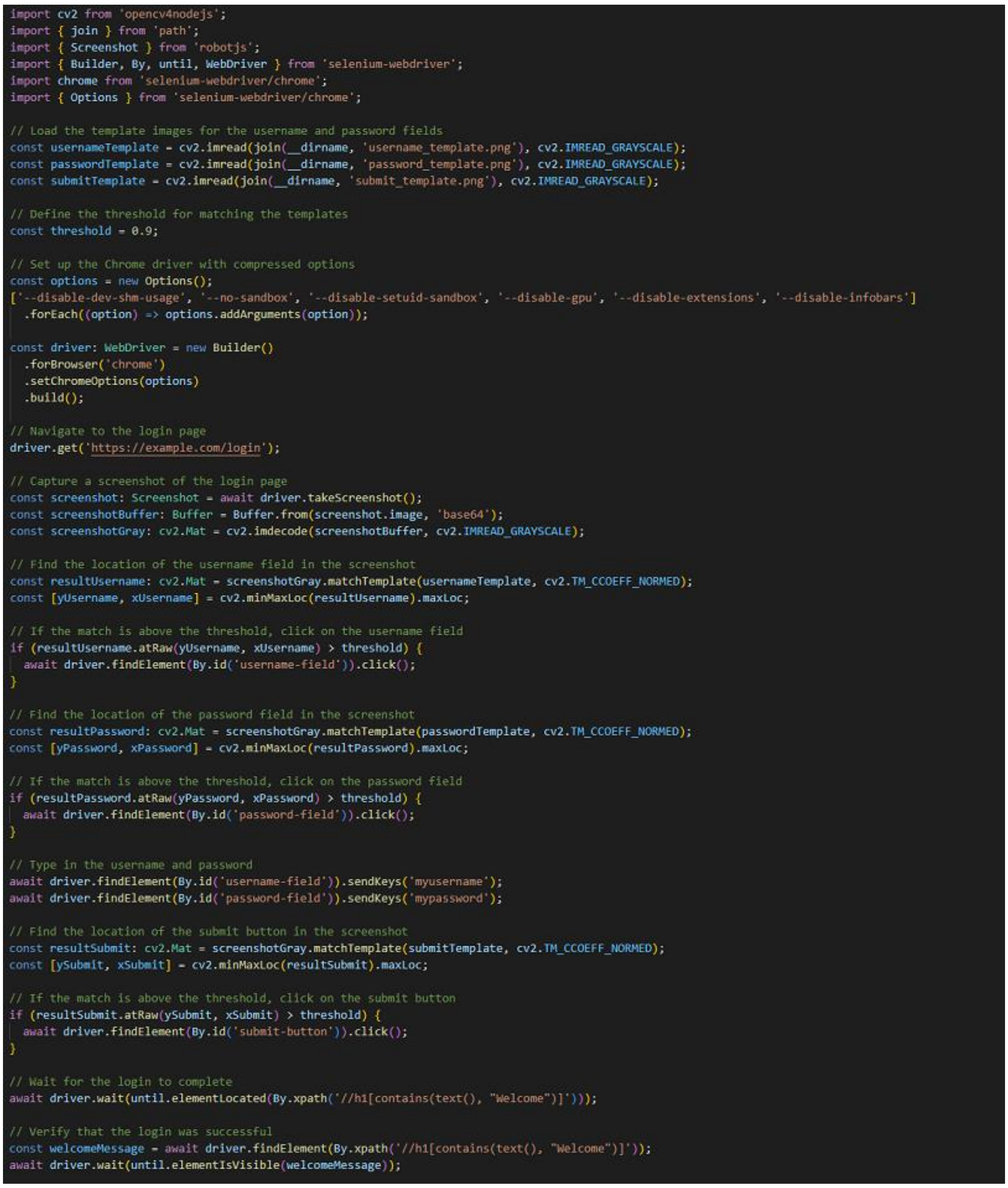


```
from sikuli import *

# if picture matchs more than 90% with capture picture
Settings.MinSimilarity = 0.9

# Capture the login form fields and the "login" button
username = capture("username.png")
password = capture("password.png")
login_button = capture("login_button.png")

# Replay the test
type(username, "johndoe")
type(password, "123456")
click(login_button)
# Wait for the login process to complete
wait(5)

# Check if the login was successful
if exists("home_page.png"):
    print("Login successful!")
else:
    print("Login failed!")
```

**Figure 8.** Simple C&R code snippet to test the login button

In this example, the login page is tested by capturing and comparing the login elements with the captured image. The next line is executed if the image matches at least 90%, similar to the target picture. The correct username and password are inserted; if the home page appears, the user is logged in successfully; otherwise, the user logged in failed.

## APPENDIX 2: Example of UI testing using Visual Localization



```typescript
import cv2 from 'opencv4nodejs';
import { join } from 'path';
import { Screenshot } from 'robotjs';
import { Builder, By, until, WebDriver } from 'selenium-webdriver';
import chrome from 'selenium-webdriver/chrome';
import { Options } from 'selenium-webdriver/chrome';

// Load the template images for the username and password fields
const usernameTemplate = cv2.imread(join(__dirname, 'username_template.png'), cv2.IMREAD_GRAYSCALE);
const passwordTemplate = cv2.imread(join(__dirname, 'password_template.png'), cv2.IMREAD_GRAYSCALE);
const submitTemplate = cv2.imread(join(__dirname, 'submit_template.png'), cv2.IMREAD_GRAYSCALE);

// Define the threshold for matching the templates
const threshold = 0.9;

// Set up the Chrome driver with compressed options
const options = new Options();
['--disable-dev-shm-usage', '--no-sandbox', '--disable-setuid-sandbox', '--disable-gpu', '--disable-extensions', '--disable-infobars']
  .forEach((option) => options.addArguments(option));

const driver: WebDriver = new Builder()
  .forBrowser('chrome')
  .setChromeOptions(options)
  .build();

// Navigate to the login page
driver.get('https://example.com/login');

// Capture a screenshot of the login page
const screenshot: Screenshot = await driver.takeScreenshot();
const screenshotBuffer: Buffer = Buffer.from(screenshot.image, 'base64');
const screenshotGray: cv2.Mat = cv2.imdecode(screenshotBuffer, cv2.IMREAD_GRAYSCALE);

// Find the location of the username field in the screenshot
const resultUsername: cv2.Mat = screenshotGray.matchTemplate(usernameTemplate, cv2.TM_CCOEFF_NORMED);
const [yUsername, xUsername] = cv2.minMaxLoc(resultUsername).maxLoc;

// If the match is above the threshold, click on the username field
if (resultUsername.atRaw(yUsername, xUsername) > threshold) {
  await driver.findElement(By.id('username-field')).click();
}

// Find the location of the password field in the screenshot
const resultPassword: cv2.Mat = screenshotGray.matchTemplate(passwordTemplate, cv2.TM_CCOEFF_NORMED);
const [yPassword, xPassword] = cv2.minMaxLoc(resultPassword).maxLoc;

// If the match is above the threshold, click on the password field
if (resultPassword.atRaw(yPassword, xPassword) > threshold) {
  await driver.findElement(By.id('password-field')).click();
}

// Type in the username and password
await driver.findElement(By.id('username-field')).sendKeys('myusername');
await driver.findElement(By.id('password-field')).sendKeys('mypassword');

// Find the location of the submit button in the screenshot
const resultSubmit: cv2.Mat = screenshotGray.matchTemplate(submitTemplate, cv2.TM_CCOEFF_NORMED);
const [ySubmit, xSubmit] = cv2.minMaxLoc(resultSubmit).maxLoc;

// If the match is above the threshold, click on the submit button
if (resultSubmit.atRaw(ySubmit, xSubmit) > threshold) {
  await driver.findElement(By.id('submit-button')).click();
}

// Wait for the login to complete
await driver.wait(until.elementLocated(By.xpath('//h1[contains(text(), "Welcome")]')));

// Verify that the login was successful
const welcomeMessage = await driver.findElement(By.xpath('//h1[contains(text(), "Welcome")]'));
await driver.wait(until.elementIsVisible(welcomeMessage));
```

**Figure 9.** Testing login page using visual localization written in TypeScript

In this example, OpenCV is used to train template images of the username field, password field, and submit button, which is used to compare to what will be captured during the test. Selenium WebDriver is used to set up the testing environment (Chrome) and interact with UI elements of the webpage. In the initial step, all trained template images and threshold are loaded and defined respectively for comparison. Chrome driver is configured with compressed options to reduce memory usage and unnecessary config. After navigating to the desired website, a screenshot of the login page is captured, and converted the resulting image to grayscale for faster processing with processed templates. Next, username and password fields are checked by comparing results from the previous step to target images if more than 90% of similarities, username, and password fields are clicked using Selenium functions. If the test passes every checkpoint before, username and password in text form with be inserted into their fields. Submit button is lastly navigated, and click if it is identical to the template. "Welcome" text is displayed as the test completely passes. Image recognition is skipped in this example because it is irrelevant to the desired study.

## APPENDIX 3: Example of UI testing using DOM-Based Localization

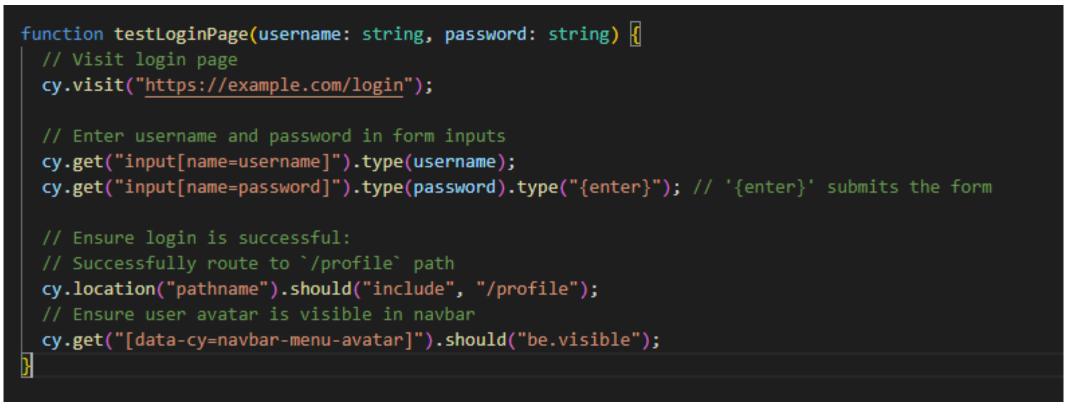
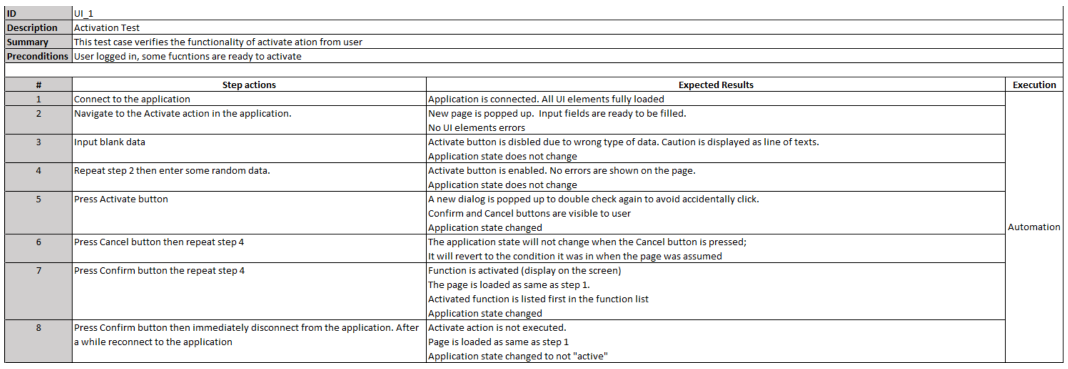

```
function testLoginPage(username: string, password: string) {
  // Visit login page
  cy.visit("https://example.com/login");

  // Enter username and password in form inputs
  cy.get("input[name=username]").type(username);
  cy.get("input[name=password]").type(password).type("{enter}"); // '{enter}' submits the form

  // Ensure login is successful:
  // Successfully route to `/profile` path
  cy.location("pathname").should("include", "/profile");
  // Ensure user avatar is visible in navbar
  cy.get("[data-cy=navbar-menu-avatar]").should("be.visible");
}
```

**Figure 10.** Test login page using Cypress

The code snippet from Figure 7 presents how the login page is tested using DOM-Based Localization using Cypress framework. After visiting the page successfully, Cypress get() function will find and compare user interface elements in the entire HTML page to desired DOM elements ("input[name=username]" and "input[name=password]"). If they match, the username and password will be filled in by the type() function. Basically, the test can end here since if an error happens in this step; a new error log will be shown. However, it could be more user-friendly if testers need to look at the log whenever executing test cases. Then DOM elements("[data-cy=navbar-menu-avatar]") are also used to check that the user avatar is visible, and the test passed.

## APPENDIX 4: Activation test specification



| | |
|---|---|
| **ID** | UI_1 |
| **Description** | Activation Test |
| **Summary** | This test case verifies the functionality of activate ation from user |
| **Preconditions** | User logged in, some fucntions are ready to activate |

| # | Step actions | Expected Results | Execution |
|---|---|---|---|
| 1 | Connect to the application | Application is connected. All UI elements fully loaded | |
| 2 | Navigate to the Activate action in the application. | New page is popped up.  Input fields are ready to be filled.<br>No UI elements errors | |
| 3 | Input blank data | Activate button is disbled due to wrong type of data. Caution is displayed as line of texts.<br>Application state does not change | |
| 4 | Repeat step 2 then enter some random data. | Activate button is enabled. No errors are shown on the page.<br>Application state does not change | |
| 5 | Press Activate button | A new dialog is popped up to double check again to avoid accidentally click.<br>Confirm and Cancel buttons are visible to user<br>Application state changed | Automation |
| 6 | Press Cancel button then repeat step 4 | The application state will not change when the Cancel button is pressed;<br>It will revert to the condition it was in when the page was assumed | |
| 7 | Press Confirm button the repeat step 4 | Function is activated (display on the screen)<br>The page is loaded as same as step 1.<br>Activated function is listed first in the function list<br>Application state changed | |
| 8 | Press Confirm button then immediately disconnect from the application. After a while reconnect to the application | Activate action is not executed.<br>Page is loaded as same as step 1<br>Application state changed to not "active" | |

**Figure 11.** Test Specification