AUTOMATED TESTING OF SYSTEMS OF SYSTEMS

A Thesis

by

Özge AKAT

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
January 2024

AUTOMATED TESTING OF SYSTEMS OF SYSTEMS

Approved by:

Prof. Dr. Hasan Sözer, Advisor
Department of Computer Science
Ozyegin University

Dr. Reyhan Aydoğan
Department of Computer Science
Ozyegin University

Dr. Yunus Emre Selçuk
Department of Computer Engineering
Yıldız Technical University

Date Approved: 5 January 2024

I dedicate this work to my family, who is always by my side with their love, support and interest, and to my dear friends who encourage me on this path. Their strong support has been the biggest driving force in the realization of this thesis. I would also like to express my gratitude to my teachers and academic professors who have guided me at every stage of my life and brought me to this point. Each of them are valuable people who form the basis of my successes behind this work.

ABSTRACT

There are various kinds of software applications like mobile and Web applications. These applications have different types of user interfaces and user interaction methods. Hence, test automation tools are either dedicated or configured for a particular kind of application. Test scenarios can be implemented in the form of scripts and test execution can be automated separately for each application type. However, there are systems of systems that embody multiple types of applications deployed on various platforms. Test scenarios might cross-cut these applications to be controlled collectively in the test script. In this thesis, we propose an approach for testing cross-platform systems of systems. We present an application of it on a real system, involving a mobile and a Web application that are supposed to work in coordination. Our approach integrates a set of existing tools to facilitate test automation. It provides testers with a unified interface for developing test scripts that involve both mobile and Web applications. We conduct an industrial case study with participants from various backgrounds and show that our tool can reduce the testing effort significantly.

ÖZETÇE

Mobil ve Web gibi çeşitli platformlar için yazılım uygulamaları geliştirilmektedir. Bu uygulamaların farklı türde kullanıcı arayüzleri ve kullanıcı etkileşim yöntemleri bulunmaktadır. Bu nedenle, test otomasyon araçlarının belirli bir uygulama türüne özel geliştirilmesi veya yapılandırılması gerekmektedir. Test senaryoları komut betimleri şeklinde tanımlanabilmekte ve her uygulama türü için ayrı ayrı otomatik olarak çalıştırılabilmektedir. Ancak, farklı platformlarda konuşlandırılan birden fazla uygulama türünü bünyesinde barındıran sistemler sistemleri bulunmaktadır. Bir sistemler sistemi için oluşturulmuş test senaryolarında yer alan test adımlarının bu farklı platformlar ve uygulamamlar üzerinde toplu olarak kontrol edilerek çalışması gerekebilmektedir. Bu tez çalışmasında, birden fazla ve farklı tipte platform kullanan sistemler sistemlerini test etmek için bir yaklaşım önerilmektedir. Önerilen yaklaşımın gerçek bir sistem üzerinde uygulaması sunulmaktadır. Bu sistemde koordineli çalışması gereken bir mobil ve bir Web uygulaması bulunmaktadır. Yaklaşımımız, test otomasyonunu kolaylaştırmak için bir dizi mevcut aracı entegre etmektedir. Ortaya çıkan araç, test uzmanlarına hem mobil hem de Web uygulamaları üzeirnde çalışabilen test komut betimlerini geliştirmek için bütünsel bir arayüz sağlamaktadır. Farklı altyapılardan gelen katılımcılarla gerçekleştirilien bir endüstriyel vaka çalışması ile, aracımızın test maliyetini önemli ölçüde azaltabildiği gösterilmektedir.

ACKNOWLEDGEMENTS

First, my advisor Prof. Dr. I would like to thank Hasan Sözer for all his help and guidance. I would also like to thank Vestel Electronics for sharing their code base and experience with me and supporting my experiments.

There are many people who support me in my efforts throughout this work. On this occasion, I would first like to express my sincere gratitude to Merve Koyuncu and Ayberk Kut, who are my biggest supporters, encouragers and sources of motivation. I also owe a special thanks to Raafeh Mahmood, who patiently answered and helped me with my questions, not only on personal but also on technical matters.

I would also like to express my gratitude to my beloved mother Gülten, my father Hayati and my sister Ezgi, who were proud of me at every stage of my education life. It would not have been possible to complete this work without their strong support and love.

I would like to express my sincere gratitude to my former manager, Alev Tezel, who supported me in starting this process.The contributions of her played an important role in the formation of this thesis. I am grateful to them for their interest, guidance and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acronyms

BDD Behavior-Driven Development. 2, 3, 4, 5, 6, 8

CPMS Charge Point Management System. ix, 2, 16, 17, 18, 30, 31, 32, 33

CPO Charging Point Operator. 17

EVC Electric Vehicle Charger. 16, 17

GUI Graphical User Interface. 7, 25

TDD Test-Driven Development. 4

VGT Visual GUI Testing. 7

XP Extreme Programming. 4

CHAPTER I

INTRODUCTION

The cost of testing activities may account for at least half of the development costs [1, 2]. The cost can further increase depending on the required reliability level and the size of the system under test. Test automation is adopted to reduce this cost [3, 4] in almost every domain including Web applications [5, 6, 7] and mobile applications [8]. These applications possess distinct user interfaces and user interaction methods. Consequently, test automation tools are either specifically designed or configured to cater to a particular type of application and platform. Test scenarios can be implemented in the form of scripts, and test execution can be automated separately for each type of application. However, certain complex systems, known as systems of systems, incorporate multiple types of applications. Usage scenarios that should be tested for these systems may cut across various platforms, and it might be necessary to control them collectively in a single test script (See Fig. 1). For instance, a user of a Web application might need to observe the result of an action that is performed by another user on a mobile application. However, there is no direct automation support for the execution of such test scripts.

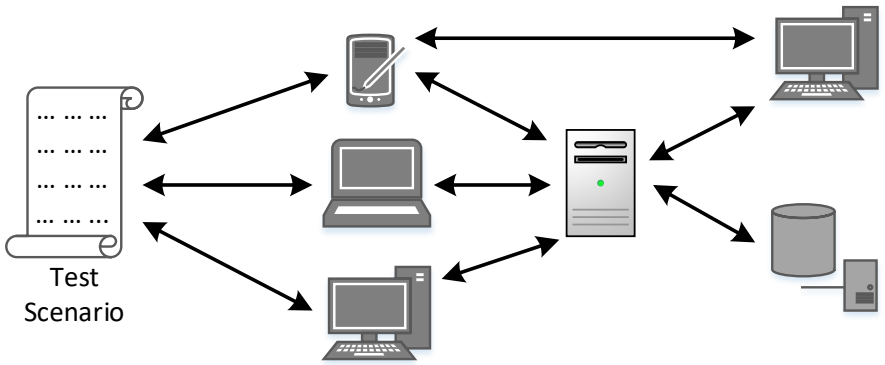

Figure 1: A test scenario that interacts with multiple cross-platform applications.

There have been studies [9, 10, 11, 12] to explore test automation strategies when dealing with multiple platforms. However, these studies focus on testing a single system that is deployed on various platforms. Their aim is to identify inconsistencies across platforms, such as cross-platform inconsistencies for mobile applications [9] and cross-browser inconsistencies for Web applications [12]. Additionally, in cases where different user interfaces are adopted for a wide range of platforms, the research goal is to reduce duplicated testing efforts for these platforms [11]. We tackle a different problem in this thesis. Our goal is to facilitate the automated testing of multiple, cross-platform systems that are supposed to work in coordination. We integrate a set of existing tools to provide testers with a unified interface for developing test scripts involving both mobile and Web applications. We facilitate test automation by following the Behavior-Driven Development (BDD) approach [13]. Test scripts can be developed as a composition of natural-language constructs that access multiple platforms concurrently. Our approach integrates a set of existing tools to facilitate test automation. It provides testers with a unified interface for developing test scripts that involve both mobile and Web applications.

We present an application of our approach on a real Charge Point Management System (CPMS) that manages the charging process for electrical vehicles. It has a Web interface for operators and mobile application for users. Operations performed by the user on the mobile application affects the web interface, or vice versa, operations on the web interface affects the mobile application. We conduct an industrial case study with 5 participants from various backgrounds and show that our tool can reduce the testing effort significantly. Overall test execution time for 5 test scenarios is reduced from 27 minutes down to 4 minutes when execution of test scenarios are automated. The time it takes to develop test scripts for test automation ranges between 3 hours and 34 hours, depending on the level of expertise and experience of the developer. Considering the reduction of 23 minutes in test execution time, the effort required for test script preparation

can be amortized after 8 to 89 test executions. We find this cost acceptable in a continuous integration environment, where the system is subject to frequent modifications, extensions and as such, repeated test executions. High frequency of test executions is common in continuous integration settings, where they are repeated at least once or even several times on a daily basis [14].

The remainder of this thesis is organized as follows. We provide background information on BDD in the following Chapter. We summarize the related studies in Chapter 3. We explain the implementation of our approach in Chapter 4. We present our experimental setup in Chapter 5. We present and discuss results in Chapter 6. Finally, we share our concluding remarks in Chapter 7.

CHAPTER II

BEHAVIOR-DRIVEN SOFTWARE DEVELOPMENT

Behavior-driven Software Development (BDD) [15] originates from Test Driven Development (TDD) [16], which was introduced as one of the practices applied in agile development processes, namely eXtreme Programming (XP) [17], in particular. TDD is also known as test-first coding or test driven design, where tests have to be provisioned before the implementation of the system. This approach forces the developer or the designer to clarify the expected behavior of the system before constructing it. Tests are automatically executable and they also serve as documentation for clarifying the expected functionality.

TDD was initially introduced as a coding practice, where the developers are expected to develop unit tests before implementing methods or classes that make these tests pass. BDD makes the same approach applicable at a higher level of abstraction, where acceptance tests can be written from the customer point of view, before the system is designed and developed. It offers the opportunity to write test scenarios in spoken language. This allows team members to easily create behavioral scenarios without requiring a technical background. Business analysts create user stories with the information they obtain from customer interviews and prepare test scenarios based on these stories. Even customers can write test scenarios. It is possible to automatically execute these scenarios on the system and test its behavior. This approach essentially reduces maintenance costs and minimizes project risks [18].

Automated execution of user-level test scenarios are supported by test tools and frameworks. Although these human-readable scenarios are written in natural language, they have to conform to certain syntax rules so that they can be interpreted by the tools and mapped to executable code. Cucumber [15, 19] is one

of the popular tools used for BDD and it requires test scenarios to be written in *Gherkin* syntax. The specification of test scenarios are supported by the use of a set of predefined keywords as listed below [19]:

- *Feature*: Specifies the feature to be tested.
- *Scenario*: Specifies the name of the scenarios to be applied within a feature.
- *Given*: Describes the initial status of the system.
- *When*: Describes an event or action.
- *And/But*: Used for specifying the (lack of) existence of one or more events together with another one that is provided as part of *Given*, *When*, *Then* specifications. *And* refers to positive events (e.g., one should see an icon on the screen), *But* refers to negative events (e.g., one should not see an icon on the screen).
- *Then*: Specifies the expected result after an event or action.

Each of these operations is called as a *Step*. An example test scenario that is written in Gherkin format is shared in Figure 2. Hereby, the scenario includes logging into the website, adding a device, and controlling that device in the mobile application.

In principle, there can be multiple scenarios defined for each feature although a single scenario is included in a feature in Figure 2. Each scenario starts with a *Given* step, where the initial state of the system is clarified. In this particular example, the Web application must be opened first. After the page loads, email and password must be entered to log in. Next, the necessary steps are defined using a series of *And* steps.

```
@tag
Feature: Operator side vs Client side
  @tag1
  Scenario: User login the web application and add device then check
   device in mobile application
    Given open web
    When click web "email"
    And type web "emailAdd" "email"
    And click web "password"
    And type web "passwordInput" "password"
    And click web "loginWeb"
    And click web "deviceManagement"
    Then click web "addChargePoint"
    And click web "deviceNameWeb"
    And type web "name" "deviceNameWeb"
    And click web "save"
    And open app
    When click app "skipApp"
    And click app "searchBar"
    Then type app "name" "searchBar"
    And the search results should contain "name"
```

Figure 2: An example implementation of a test scenario in Gherkin format.

Our solution approach relies on BDD and in particular the Cucumber framework, which automates the execution of test scenarios that are specified in structured natural language as shown in Figure 2. These scenarios can also be parameterized and can be executed multiple times with varying inputs. This approach allows one to define and read test scenarios more easily and quickly without any technical background. However, the current tools and frameworks do not support the development of test scenarios that involve interactions with multiple platforms at the same time. We summarize the related studies and position our work in the following chapter from this perspective.

CHAPTER III

RELATED WORK

The behavior and the Graphical User Interface (GUI) of a software application might change depending on the deployed platform. Research efforts so far focus on the identification of these platform inconsistencies, especially for Web applications [12] and mobile applications [9] that can be deployed on a variety of platforms. In this work, we focus on the testing of systems of systems, where each involved system is deployed on a different platform.

The increasing number and variety of platforms increase test automation efforts as well [11]. Frameworks like Apache Cordova, Xamarin, and React Native enable the development of a single application that can be deployed on multiple platforms. However, a separate test script has to be developed for testing the application on each platform. x-PATeSCO [20] automatically generates test scripts for multiple platforms. However, it focuses on testing a single application at a time, unlike our approach.

Test scripts can be fragile due to changes in the GUI layout or code, as well as differences among the deployed platforms. Visual GUI Testing (VGT) [21] gained popularity due to its adaptability and resilience to these changes and differences. However, VGT approaches proposed so far focus on testing a single application rather than multiple types of applications on various platforms at the same time.

Appium [22] supports test automation across many platforms covering mobile (iOS, Android, Tizen), browser (Chrome, Firefox, Safari), desktop (macOS, Windows), and TV (Roku, tvOS, Android TV, Samsung) applications. It also supports several programming languages (e.g., Java, Pyhton, Ruby) for developing test scripts. However, to use Appium, one needs to set up a set of drivers and plugins depending on the target platform, and programming experience is

necessary to develop test scripts. Our goal is to provide support test automation for multiple platforms at the same time and at a hig level of abstraction by following BDD principles [13]. With our approach, test scripts can be written using structured natural language constructs (i.e., in Gherkin format) that access multiple platforms concurrently, without worrying about the underlying setup. In the following chapter, we discuss our approach and its implementation in detail.

CHAPTER IV

SYSTEMS OF SYSTEMS TESTING APPROACH

An overview of our approach is depicted in Figure 3, where a set of tools are integrated. In particular, our approach employs Cucumber [13], Selenium [23] and Appium [22] for enabling test automation. A Selenium driver is used for steering a Web application on a Chrome browser. An Appium driver is used for controlling a mobile application that works on an Android phone. Cucumber is used for developing test scenarios. The test steps listed in these scenarios are mapped to executable test scripts by test fixtures that we developed in Java.

Test scenarios are developed in Gherkin format, where test steps are specified in structured natural language. Each of these test steps are mapped to a test fixture by Cucumber. Test steps are annotated to differentiate among a number of target applications and platforms. Text fixtures use these annotations to execute the intended user actions by employing the corresponding driver and interface. The implementation of the test fixtures is available online at a public repository. While our current implementation utilizes Selenium and Appium, it can be enhanced with alternative tools to accommodate diverse platforms. We used the Bridge Pattern [24] for abstracting away platform details from the set of possible generic test steps.

We introduce a configuration file so that users can easily change data regarding

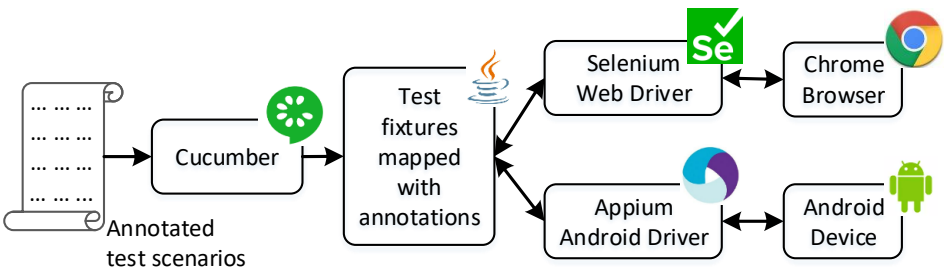

Figure 3: Implementation of the approach with an integrated set of tools.

the test setup such as device information, button information, and URL information. When a change needs to be made, editing the content of this configuration file will be sufficient. In order to create this configuration file, it is first necessary to obtain the code equivalents of the data we use in the scenarios. To easily find the names of these variables, the recorder feature of Appium and Selenium is used. In the Appium example, users must first start the Appium server and pre-define the dependencies of the device on which they will use the Appium Inspector. Performing these steps is detailed through an example configuration:

```
"appium:deviceName": "sdk gphone64 arm64",
"platformName": "Android",
"appium:platformVersion": "14",
"appium:appPackage": "com.google.android.x",
"appium:appActivity": "com.google.android.x.HomeActivity"
```

This specified configuration targets a specific Android device or emulator. Appium Inspector provides users with the opportunity to perform various operations by presenting the screen of the specified device or emulator with a visual interface. These definitions will be used in the configuration file. Through Appium Inspector, users can perform actions such as clicking on items, viewing item locations, and taking screenshots. Additionally, code snippets obtained through Inspector can be used to develop test automation scenarios.

Figure 4 in the following page presents a code fragment that belongs to the test scenario recorded using the Appium inspector. The IDs that used in this code fragment must be consistent with those that are used as part of the configuration file prepared for the system under test.

```
@Test
public void sampleTest() {
 WebElement el2 = driver.findElementByAccessibilityId("Home");
 el2.click();
 WebElement el4 = driver.findElementByClassName("android.view.
ViewGroup");
 el4.click();
 WebElement el5 = driver.findElementById("com.surgemobile.vec.
android:id/skipTV");
 el5.click();
 WebElement el7 = driver.findElementById("com.surgemobile.vec.android
/chargingPointsSearchET");
 el7.click();
 WebElement el8 = driver.findElementById("com.surgemobile.vec.android
/chargingPointsSearchET");
 el8.sendKeys("DV–");
 el8.click();
 WebElement el9 = driver.findElementByXPath("android.widget.
RelativeLayout[1]/android.view.ViewGroup");
 el9.click();
    }
```

Figure 4: A sample snippet from the test fixture code from Appium inspector.

On the Selenium side, the Chrome plug-in Selenium IDE recorder is used. The code fragment shown on the following page in Figure 5 belongs to the test scenario recorded using Selenium Recorder. The IDs in this code snippet will be used in the configuration file of the tested system as well.

```
@Test
public void VEC() {
 driver.get("https://vesteuwebapp.azurewebsites.net/");
 driver.manage().window().setSize(new Dimension(1440, 807));
 driver.findElement(By.id("logonIdentifier")).click();
 driver.findElement(By.id("logonIdentifier")).sendKeys
("ozge@vestel.com.tr");
 driver.findElement(By.id("password")).click();
 driver.findElement(By.id("password")).sendKeys("asd.12");
 driver.findElement(By.id("next")).click();
 driver.findElement(By.linkText("User Management")).click();
 driver.findElement(By.cssSelector("#UserManagementGrid > .k-loading
-image")).click();
 driver.findElement(By.id("FieldFilter")).click();
 WebElement element = driver.findElement(By.linkText
("Company name"));
 driver.findElement(By.id("FieldFilter")).sendKeys("Ozge");
 driver.findElement(By.linkText("ozge Test")).click();
```

Figure 5: A sliced sample snippet from the test fixture code from Selenium.

The found IDs are saved in the configuration file in JSON format [25]. In order to avoid using ID hard-coded in the Cucumber scenario, the variable name assigned to the ID is determined.

```
"skipApp":"com.surgemobile.vec.android:id/skipTV",

"searchArea":"chargingPointsSearchET",

"url":"https://vesteuwebapp.azurewebsites.net",

"loginEmail":"logonIdentifier",

"password":"password"
```

Finally, scripts should be written by Cucumber as shown in Figure 6 on the next page.

```
1  @tag1
2  Scenario: User login the web application and add device then check
3  device in mobile application
4      Given open web
5      When click web "email"
6      And type web "emailAdd" "email"
7      And click web "password"
8      And type web "passwordInput" "password"
9      And click web "loginWeb"
10     And open app
11     When click app "skipApp"
12     And click app "searchBar"
13     Then type app "name" "searchBar"
14     And the search results should contain "name"
```

Figure 6: The implementation of a sample scenario in Gherkin format for automated execution with Cucumber.

In the above scenario, a login operation is performed on a Web page first. Then, a search operation is performed on a mobile application. These operations are automatically executed by Cucumber by mapping each test step to executable Java source code, which are called as *test fixtures*. These test fixtures should be developed only once for each test step. However, test steps can be parameterized and reused in many different test scenarios. A sample snippet from the test fixture that corresponds to the scenario above is shared in Figure 7 on the next page. Hereby, lines $1-3$ correspond to the test setup, where drivers are initialized. Lines $5-10$ correspond to the first line in the scenario listed above. Lines $10-17$ correspond to the 12-14th lines, where a charger is searched.

```
public class StepDefinition {
        public AppiumTest DriverA;
        public SeleniumTest WebTest;

        @Given("open web")
        public void open_web() {
                WebTest = new SeleniumTest();
        }
        @Given("open app")
        public void open_app() {
                DriverA = new AppiumTest();
        }
    @Then("type app {string} {string}")
        public void type_app(String input2,String buton){
                DriverA.type(input2,buton);
        }
```

Figure 7: A sliced sample snippet from the test fixture code.

A complete user manual for the test tool is provided in Appendix C. In short, the usage of the tool involves the following 5 steps: *i)* test steps that are related to the mobile application are manually applied and recorded using the Appium Inspector to collect IDs necessary to execute these steps, *ii)* test steps that are related to the Web application are manually applied and recorded using the Selenium IDE to collect IDs necessary to execute these steps, *iii)* The collected IDs during the previous steps are defined in a JSON configuration file, *iv)* Test scenarios are developed in Gherkin format based on the created configuration file, and *v)* The developed test scenarios are executed with the Cucumber framework. These steps are defined assuming that the test scenarios involve a mobile application and a Web application. Hence, there are only two steps at the beginning to collect IDs. The number of steps can be increased as we increase the number of types of platforms involved in the test scenarios. The last 3 steps remain the same, regardless of the number of platforms involved.

We evaluate our approach with an industrial case study. We describe our experimental setup in the following chapter.

CHAPTER V

EXPERIMENTAL EVALUATION

In this chapter, we describe the industrial case study that is used for evaluating our approach. In the following section, we specify our research questions. Then, in Section 5.2, we introduce the profile of participants who are involved in our case study. We explain the system of systems that is used as the experimental object in Section 5.3. Then, in Section 5.2, we provide information regarding test scenarios that are developed for this system.

5.1 Research Questions

The goal of our evaluation is to investigate the impact of our approach on the testing effort. This impact is twofold, involving both a benefit and a cost. On one hand, we would like to know the amount of testing time that can be saved with test automation by using test scripts for testing systems of systems. On the other hand, we are also interested in the effort that is required to develop these test scripts. Accordingly, we specify the following two research questions that are aligned with our concerns.

- **RQ1.** To what extent the testing time can be reduced with test automation in comparison with manual testing for systems of systems?
- **RQ2.** How much effort is required to develop test scripts for automating the execution of test scenarios in a system of systems context?

In the following sections, we explain the experimental setup that we used for answering these questions.

5.2 Experimental Subjects

Table 1 lists the 5 participants involved in our case study. These participants are selected from 5 different companies operating in various sectors. The first column enumerates them from A to E. The second column lists the job titles of the participants. We can see that 4 of these job titles refer to engineering/developer positions that require a university degree. There is also a test technician who graduated from a two-year training program. The last column lists the years of experience for each participant.

Table 1: Profile of participants involved in the case study.

Participant ID	Job Title	Years of Experience
A	Senior Test Engineer	6
B	Full-stack Developer	7
C	Mid Test Engineer	4
D	Test Technician	3
E	Junior Test Engineer	2

At the beginning of the experiment, an on-boarding process was applied to these participants and the system to be tested was explained to them in detail. This process was undertaken to ensure that participants had a full understanding of the project's requirements, tools used, and expected outcomes. These participants were asked to describe 5 different scenarios in accordance with the main objectives of the experiment. These scenarios are explained in the last section of this chapter. We explain our experimental object in the following section.

5.3 Experimental Object

Our approach is motivated by the testing processes of a system called Charge Point Management System (CPMS). It is developed by a multi-industry manufacturer, Vestel, for managing Electric Vehicle Chargers (EVCs)[1]. CPMS is composed of 3 systems. There is a backend system that resides on the Azure cloud platform

[1] `http://vestelinternational.com/en/ev-charging-stations`

to keep track of EVC resources and the relevant IoT protocols. There is a Web application used by Charging Point Operators (CPOs) to control the charging process, EVCs and users. There is also a mobile application used by electrical vehicle owners for interacting with an EVC.

A user can start the charging process via the mobile application with or without registration. After the charging process is completed, the payment is made via Internet banking and checked through the CPO Web portal. The charging process can be initiated by using QR codes via the mobile application. Registered users can make reservations through the mobile application, depending on the availability of the connectors of EVCs. Reservations can be canceled via the mobile application or the CPO Web portal. New charging points can be added by CPOs, and users can locate these charging points on the map. When a faulty charging point is to be removed from use, it can be deleted by CPOs, so that users cannot access this device via the mobile application anymore.

CPMS has been subject to testing at various levels, including unit tests, integration tests, and system tests. These tests could be automated for a single system. For example, executable test scripts could be developed with Selenium for the CPO Web portal. However, programming expertise is required and it was not possible to automate end-to-end tests that involve multiple systems at the same time. For instance, when a user reserves a particular EVC from the mobile application, a status update should be visible at the CPO Web portal. Likewise, when a new EVC is introduced by a CPO, it should be visible on the map presented by the mobile application. Our approach facilitated the specification and automated execution of such scenarios, without requiring programming expertise.

Appendix A provides screenshots from both the Web application and the mobile application that are part of the CPMS system. Test scenarios and text fixtures that enable the automated execution of these scenarios are provided in Appendix B. We explain 5 test scenarios developed for CPMS in the following section. Our participants were asked to specify these scenarios to be used with our tool.

```
Scenario: User login the web application and add device then
check device in mobile application
Given open web
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And click web "deviceManagement"
Then click web "addChargePoint"
And click web "deviceNameWeb"
And type web "name" "deviceNameWeb"
And click web "save"
And open app
When click app "skipApp"
And click app "searchBar"
Then type app "name" "searchBar"
And the search results should contain "name"
```

Figure 8: A scenario for finding a charger.

5.4 Test Scenarios

We reviewed the manually applied test scenarios to be used for our case study. They reflect the actual usage scenarios of CPMS that are applied by testers in a controlled environment. We selected 9 most typical scenarios for automation. We further selected 5 most frequently used scenarios out of this 9 and explained them to our participants. Then, we asked them to specify these scenarios, as listed below, for their automated execution with our tool. The scenario in Figure 8 simulates a user logging into the Web app, adding a device, and then controlling that device in the mobile app. First, it opens the website; enters information in the "email" and "password" fields on the Web page; clicks the relevant button to log in; clicks on the "deviceManagement" option on the web page; clicks on the "addChargePoint" button to add a device; clicks on the "deviceNameWeb" button to determine the device name; writes in the "name" field to write the device name; clicks on the "save" button to save the device information; opens the mobile application; clicks on "skipApp"; clicks on "searchBar"; types a name in the search bar; checks that search results should include that name. This scenario simulates

```
Scenario: Reserved Charger
Given open web
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And click web "deviceManagement"
And click web "searchDeviceWeb"
And type web "deviceNameWeb" "searchDeviceWeb"
And the search results should contain "deviceNameWeb"
And click web "deviceArea"
Then check device status web
And open app
When click app "skipApp"
And click app "searchBar"
Then type app "name" "searchBar"
And the search results should contain "name"
And click app "reserved"
And click web "deviceManagement"
And click web "searchDeviceWeb"
And type web "deviceNameWeb" "searchDeviceWeb"
Then check device status web
```

Figure 9: A scenario for reserving a charger.

the user logging into the web application, adding a device, and then controlling that device via the mobile application.

This scenario in Figure 9 consists of a series of steps that simulate reserving a charger. It opens the website; enters the information in the "email" and "password" fields on the web page; clicks the relevant button to log in; clicks the "deviceManagement" option on the Web page; clicks the "searchDeviceWeb" button for device searches; writes the device name in the search box and checks that the search results should include this device name; selects the area where the specified device is located; performs the "check device status web" step to check the status of the device; opens the mobile application; clicks "skipApp"; clicks "searchBar"; types a name in the search bar; checks that search results should include that name; clicks the "reserved" button to reserve a specific charger; clicks "deviceManagement" on the Web page; searches for a device; clicks the "searchDeviceWeb"

```
Scenario: Check payment
Given open web
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And open app
When click app "skipApp"
And click app "searchBar"
Then type app "name" "searchBar"
And the search results should contain "name"
And click app "startCharge"
And click web "userManagement"
And click web "searchUser"
And type web "userName" "searchUser"
And click web "user"
Then check user status web
```

Figure 10: A scenario for checking the payment.

button for operations; writes the device name in the search box and performs the "check device status web" step to check the status of the device. This scenario simulates the process of reserving a charger using the Web and mobile interfaces. User performs verification steps before and after booking by checking device status.

This scenario in Figure 10 consists of a series of steps that simulate checking a payment. First, it opens the website; enters the information in the "email" and "password" fields on the Web page; clicks the relevant button to log in; opens the mobile application; logs in as a guest by clicking the "skipApp" option in the application; clicks the "searchBar" item; gives a name to the search bar; checks that the search results should include this name; clicks on the "startCharge" button after selecting a specific user; clicks on the "userManagement" option on the Web page; clicks on the "searchUser" button for search operations; writes the user name in the search box and returns the results with the relevant user; clicks on the "user" button to check the user's status; performs the "check user status web" step to check the user's status. Based on the user's status, it is understood whether

```
Scenario: Activate user
Given open app
When click app "email"
And type app "emailAdd" "email"
And click app "password"
And type app "passwordInput" "password"
And click app "loginWeb"
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And click web "userManagement"
And click web "searchUser"
And type web "userName" "searchUser"
And click web "user"
Then check user status web
```

Figure 11: A scenario of activating a user.

the payment transaction has been made or not. This scenario includes payment control and user management processes using the Web and mobile interfaces. This scenario in Figure 11 consists of a series of steps that simulate the activation of a user. It opens the mobile application; enters information in the "email" and "password" fields within the application; clicks the relevant button to log in; enters information in the "email" and "password" fields on the Web page; clicks the relevant button to log in; clicks "userManagement" on the Web page; clicks the "searchUser" button for search operations; writes his username in the search box and clicks the "user" button to check the results containing the relevant user; performs the "check user status web" step to check the user's status. This scenario simulates the user signing up to the mobile application and activating the user via the Web interface.

This scenario in Figure 12 consists of a series of steps to control a charger. It opens a given website; enters information in the "email" and "password" fields on the Web page; clicks the relevant button to log in; clicks on the "deviceManagement" option on the home page; clicks the "searchDeviceWeb" button for device searches; writes the device name in the search box and checks that the search

```
Scenario: Check charger
Given open web
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And click web "deviceManagement"
And click web "searchDeviceWeb"
And type web "deviceNameWeb" "searchDeviceWeb"
And the search results should contain "deviceNameWeb"
And click web "deviceArea"
And click web "deleteButton"
And click web "deviceManagement"
And click web "searchDeviceWeb"
And type web "deviceNameWeb" "searchDeviceWeb"
But the search results should contain "deviceNameWeb"
When open app
And click app "skipApp"
And click app "searchBar"
Then type app "name" "searchBar"
But the search results should contain "name"
```

Figure 12: A scenario for checking a charger.

results should include this device name; selects the area where the specified device is located; clicks on the "deleteButton" button to delete the device; returns to the device management page; opens a mobile application; skips the login steps by clicking on the "skipApp" option; clicks on the "searchBar" item on the home page; writes a name in the search bar; checks that the search results do not contain this name. This scenario simulates controlling a charger both via the Web interface and the mobile app.

Participants were informed about all the 5 scenarios described above. Then, they were asked to developed the corresponding test scripts and create the necessary configuration files to make them automatically executable. We recorded the time it takes for them to manage these tasks. We discuss the obtained results in the following chapter.

CHAPTER VI

RESULTS AND DISCUSSION

Table 2 lists the properties of the implemented test scenarios that are described in the previous chapter, the time it takes to execute them manually and their execution time with our approach. We observe an order of magnitude reduction in testing time.

Table 2: 5 test scenarios developed by participants and their execution times (in minutes) before and after automation.

Test scenario	**# of test steps**	**Manual testing time**	**Test execution time**
S1	16	7	0.83
S2	22	6	0.96
S3	17	5	0.54
S4	16	5	0.72
S5	21	4	0.81
Total	92	27	3.86

Every execution of the listed test scenarios saves more than 20 minutes time without requiring any manual effort. However, the development of test scenarios requires manual effort. Therefore, we separately asked all the 5 participants to develop these 5 test scenarios. We recorded the time they spent in various tasks to be able to develop test scenarios. Results are listed in Table 3.

Table 3: Time spent (in hours) by 5 participants on various tasks necessary for developing test scenarios.

Participant ID	**Locating IDs**	**Creating Configuration**	**Developing Test Scripts**	**Total**
A	3	1	1	5
B	1	1	1	3
C	2.8	1	1	3.8
D	16	3	5	24
E	16	2	16	34

The first column of Table 3 lists the participants. These participants must first find the IDs of all the GUI elements (e.g., buttons, text fields) to be interacted with. The second column lists the time spent for this task. The third column shows the time spent for creating a configuration file in JSON format, where the located IDs are saved to be used by test fixtures. The fourth column lists the time required by participants to develop test scripts including the time it takes to create test fixtures in Java and the test scenarios in Gherkin format.

We further continued our study with participant C, who developed 4 more test scenarios, facilitating the automation of 9 test scenarios in total. Table 4 lists the overall results obtained.

Table 4: Test scenarios and their execution time before and after automation and the test development time recorded with participant C for 9 test scenarios.

Test scenario	# of test steps	Manual testing time (min)	Test execution time (min)	Test development time (min)	Min.# of test executions for cost amortization
S1	16	7	0.83	7.12	1
S2	22	6	0.96	90	18
S3	17	5	0.54	30	7
S4	16	5	0.72	19.48	5
S5	21	4	0.81	90	28
S6	10	2	0.16	19.48	11
S7	11	3	0.23	60	22
S8	10	2	0.15	30	16
S9	15	4	0.29	19.48	5
Total	138	38	4.69	365.56	113

The first 4 columns of Table 4 are the same as those defined in Table 2. Table 4 includes two additional column at the end that lists the time it took for participant C to develop each of the 9 test scenarios and the number of test execution for cost amortization. We discuss our results and threats to their validity in the following sections.

6.1 Discussion

The efficiency gains achieved with automation are highlighted by a dramatic reduction in notable testing time. While the manual testing time of 9 scenarios

consisting of 138 steps is 38 minutes, the same tests with automation created using Cucumber are completed in 4.69 minutes. This significant time saving plays a critical role in demonstrating the practicality and advantages of the proposed approach.

We observed that the overall test execution time is reduced from 27 minutes down to 4 minutes when execution of 5 test scenarios are automated. The time it takes to develop test scripts for these scenarios ranges between 3 hours and 34 hours, depending on the level of expertise and experience of the participant. Considering the reduction of 23 minutes in test execution time, the effort required for test script preparation can be amortized after 8 to 89 test executions. This cost is negligible for continuous integration environments, where high frequency of test executions is common [14].

Test development efforts required to implement this system are noted for different roles and experience levels (See Table 1). Table 3 shows a high variation in that respect, where the development time ranges between 3 and 34 hours. A further interesting observation is that participant D required 10 hours less time than participant E to develop the test scenarios. Participant D is a technician whereas participant E is an engineer. On the other hand, participant D has one year more experience. This shows that technical background does not impact the required effort significantly but the year of experience might do so. This observation is also aligned with previously reported empirical results [26], where experience level turns out to be a more critical factor in test effectiveness compared to the education level.

It took 6 hours to develop the overall framework that integrates a set of tools. However, this is a one time investment that is agnostic to the system under test. The framework allows repeatable testing processes to be carried out more quickly and effectively. Considering the initial cost of automation, the 6-hour investment becomes more tolerable when considering the long-term benefit. Automation provides time and cost advantages in repetitive testing processes and large-scale

projects. Long-term use offsets the initial cost of automation and provides cost savings. We discuss threats to validity in the following section.

6.2 Threats to Validity

Our evaluation is subject to an *external validity* threat since we conducted a case study with a single experimental object in the context of a single company and application domain. The case study can be replicated with more experimental objects from different domains to improve the generalizability of the results.

It is very hard to find qualified volunteers to allocate hours (or even days in certain cases) for experimental evaluations. We could only work with 5 participants to evaluate the effectiveness of our approach, which is not enough for conducting analysis regarding the statistical significance of results. Hence, a *conclusion validity* threat exists due to a small group of participants involved in our case study. We diversified the profile of participants as much as possible to mitigate this threat. Each participant works in a different company and they all have varying education and experience levels.

There also exist *internal validity* threats since there are many factors that are not under our control and these factors might have impacted the results. Participants were provided with the user manual of our tool and the necessary information for developing test scenarios. However, we did not monitor their way of working. All the tasks were completed offline and not necessarily within a single time interval. Participants continued to work on the tasks whenever they had time and recorded the time they spent themselves. So, we do not know if they attained full focus and efficiency for these tasks. There might also be inaccuracies regarding the time recordings.

We do not foresee a *construct validity* threat in our evaluation. All our measurements are intended to capture effort. Effort is commonly measured in terms of time it takes to perform a task although the measurement accuracy might be a concern. We conclude the thesis in the following chapter.

CHAPTER VII

CONCLUSION AND FUTURE WORK

We introduced an approach for testing systems of systems. We integrated a set of existing tools for implementing a generic approach. Our approach provides testers with a unified interface for developing test scripts that involve multiple applications deployed on various platforms at the same time. Currently it supports Web and Android applications that are supposed to work in coordination. We also illustrated an application of our approach for the test automation of a real system that is composed of a Web and a mobile application. We conducted an industrial case study with 5 participants from various backgrounds. We observed significant effort reduction for tests that were previously performed manually. The effort required for the development of test scripts can be amortized after less than a hundred test executions. We also observed a high variation in this effort, which turned out not to be related to the educational and technical background.

As future work, we plan to increase the variety of platforms supported. We also plan to conduct additional case studies and experiments for measuring the effectiveness of our approach in test automation and effort reduction. In addition, addressing any of the validity threats explained in the previous chapter can be considered as a future work for this study.

APPENDIX A

SAMPLE SCREENSHOTS

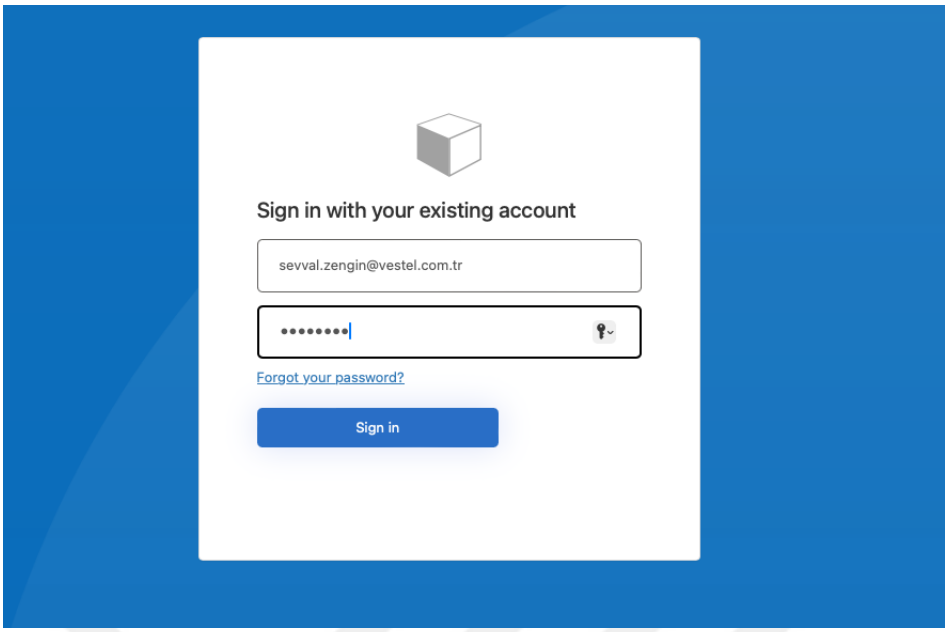

Figure 13: CPMS Website Login page.

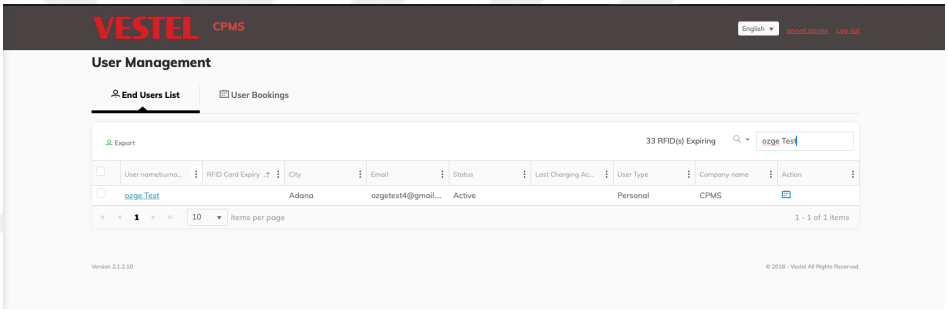

Figure 14: CPMS Website User Management page.

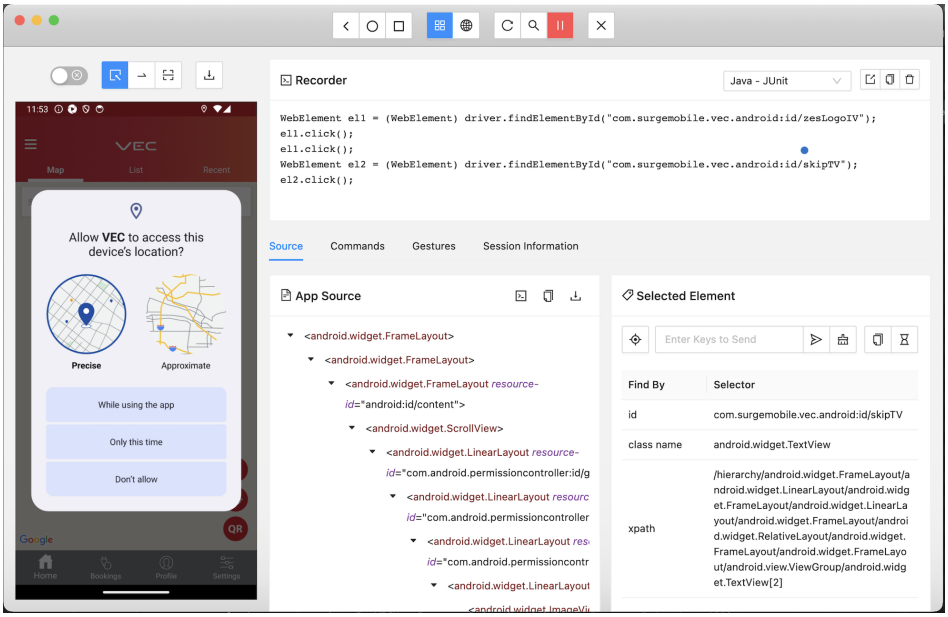

Figure 15: A screenshot from the Appium Inspector tool.

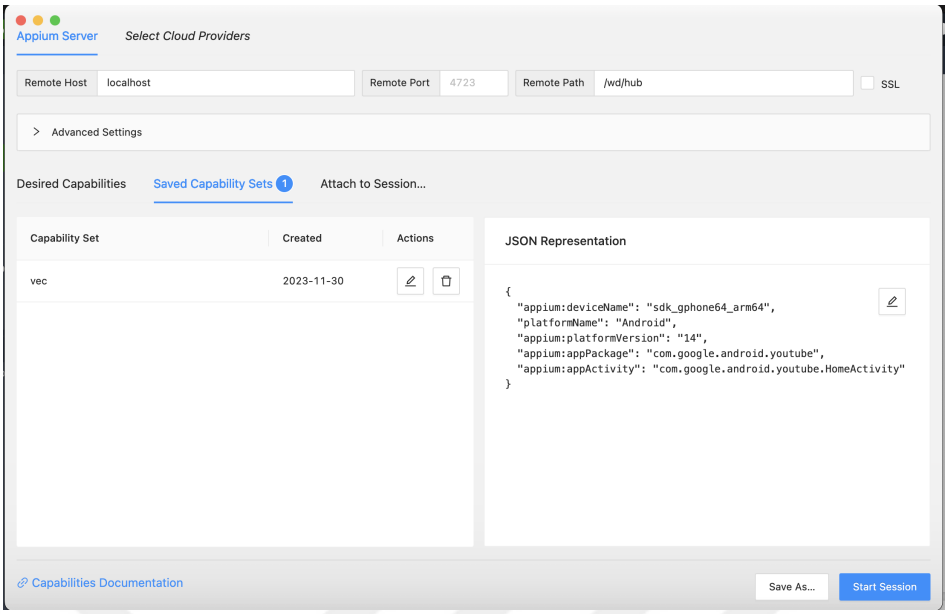

Figure 16: A sample configuration for Appium Inspector Capability Set.

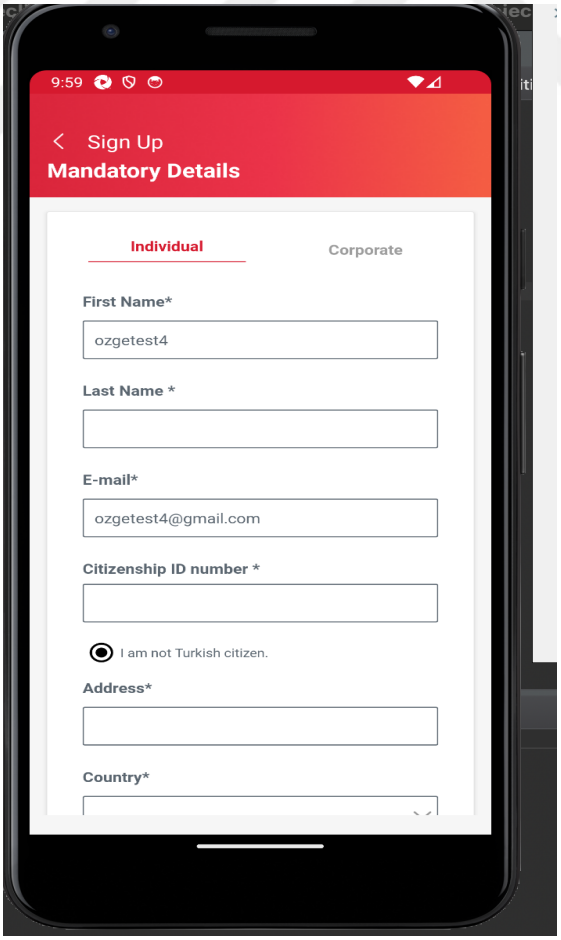

Figure 17: CPMS Mobile Application, Sign Up page.

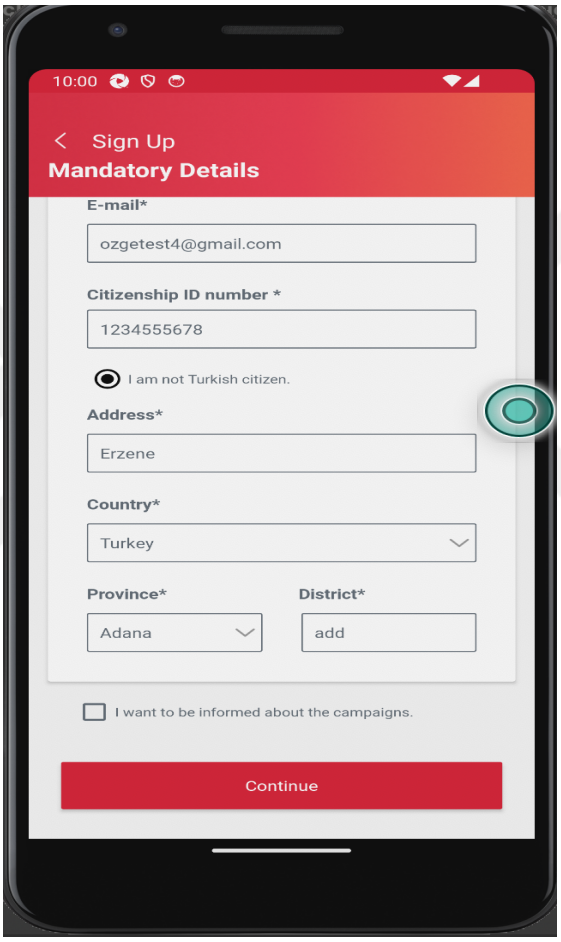

Figure 18: A screenshot taken while data inputs being entered automatically on CPMS Mobile Application Sign Up page.

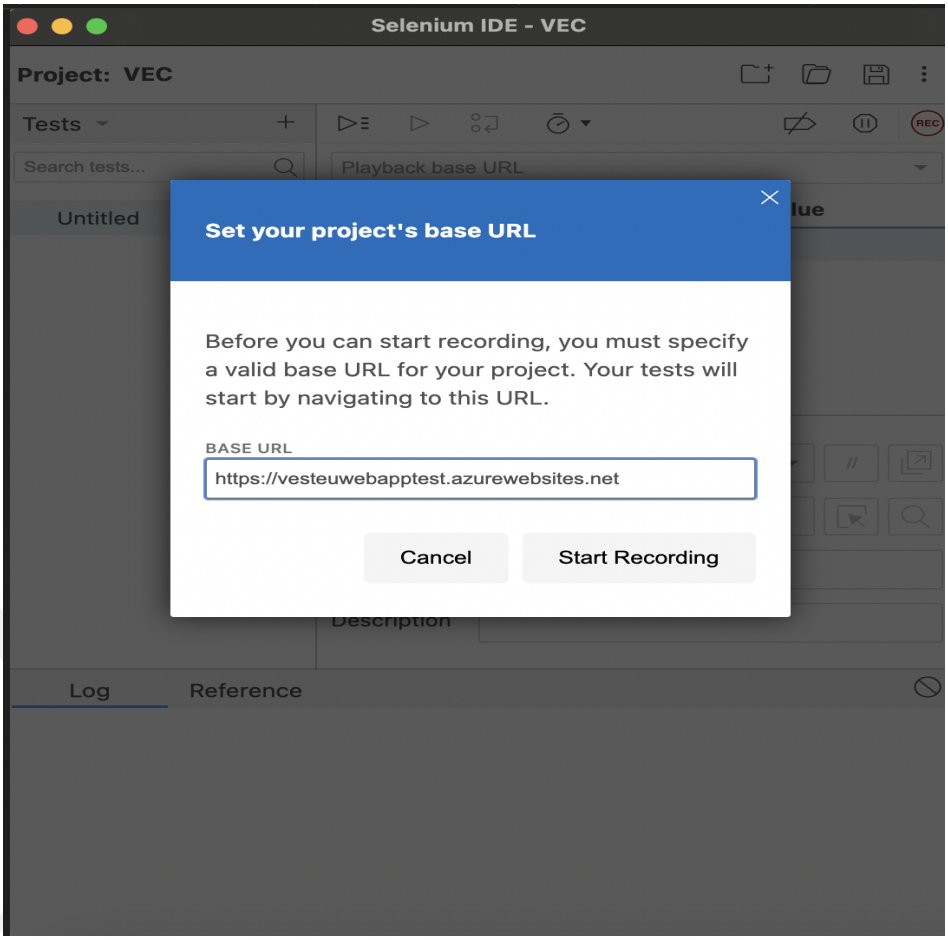

Figure 19: Using Selenium IDE for recording user actions performed after visiting a URL.

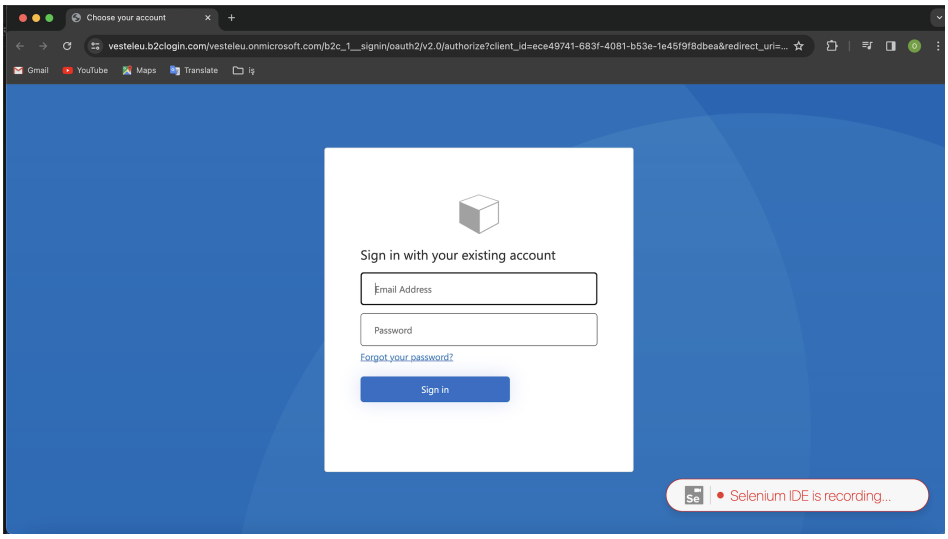

Figure 20: Selenium IDE Recorder in action while using CPMS Web interface.

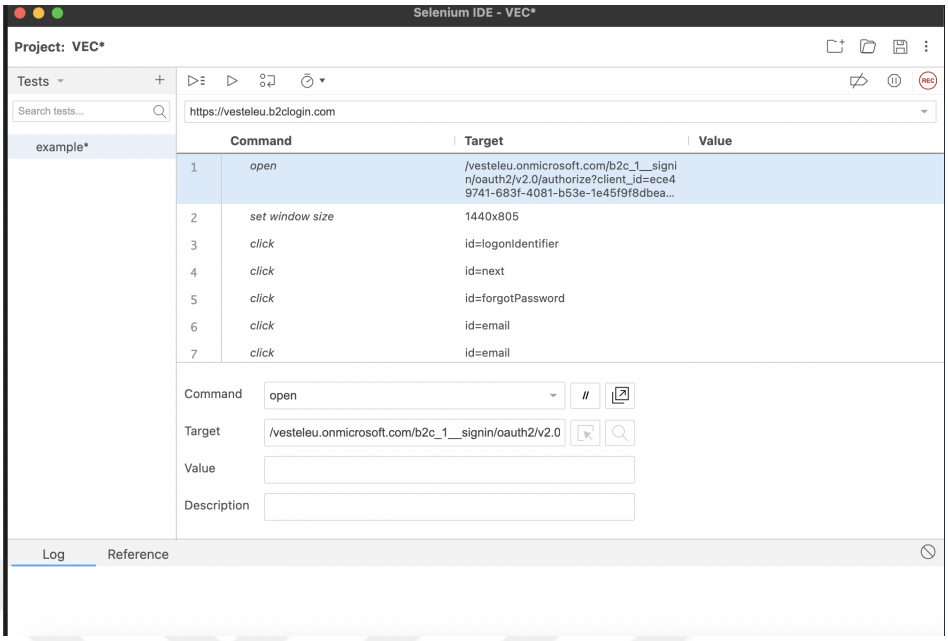

Figure 21: Sample results provided by Selenium IDE recorder after a manual test session.

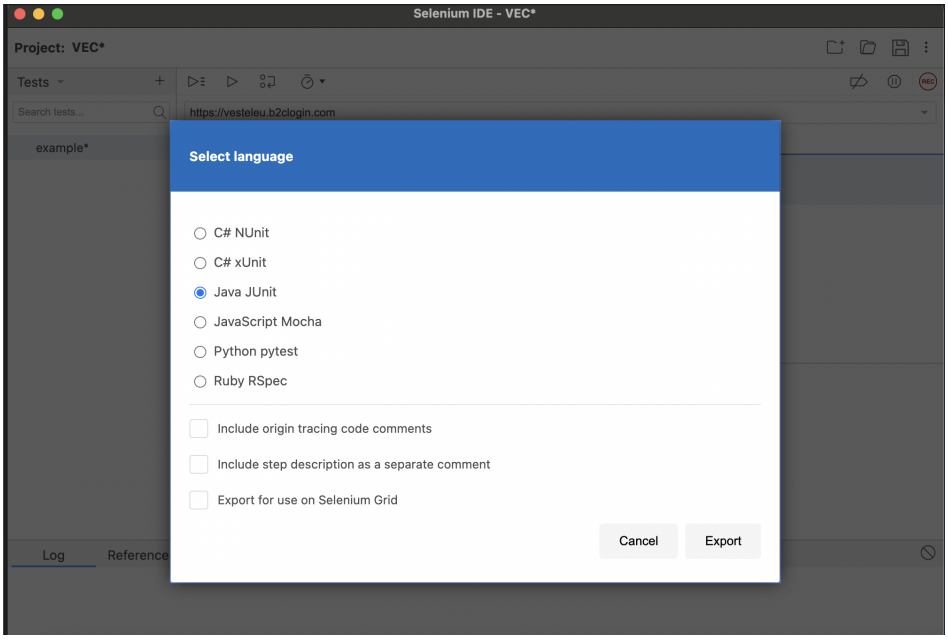

Figure 22: Exporting code from a test session performed with Selenium IDE.

APPENDIX B

TEST SCENARIOS

```
public class StepDefinition {
        public AppiumTest DriverA;
        public SeleniumTest WebTest;

        @Given("open web")
        public void open_web(){
                WebTest = new SeleniumTest();

        }
        @Given("open app")
        public void open_app() {
                DriverA = new AppiumTest();

        }
        @Given("click web {string}")
        public void click_web(String buton){
                        WebTest.clicks(buton);
                        }

        @Given("click app {string}")
        public void click_app(String buton) {
                        DriverA.clicks(buton);
                }

        @Then("type web {string} {string}")
        public void type_web(String input, String buton){
                WebTest.type(input,buton);
        }

        @Then("type app {string} {string}")
        public void type_app(String input2,String buton) {
                DriverA.type(input2,buton);
        }

        @Then("the search results should contain {string}")
        public void the_search_results_should_contain(String string){
                //check data
        }

        }
```

Figure 23: A sliced sample snippet from the test fixture code.

```
@tag
Feature: Operator side vs Client side
  @tag1
  Scenario: User login the web application
  and add device then check device
  in mobile application
    Given open web
    When click web "email"
    And type web "emailAdd" "email"
    And click web "password"
    And type web "passwordInput" "password"
    And click web "loginWeb"
    And click web "deviceManagement"
    Then click web "addChargePoint"
    And click web "deviceNameWeb"
    And type web "name" "deviceNameWeb"
    And click web "save"
    And open app
    When click app "skipApp"
    And click app "searchBar"
    Then type app "name" "searchBar"
    And the search results should contain "name"

  @tag2
      Scenario: Reserved Charger
    Given open web
    When click web "email"
    And type web "emailAdd" "email"
    And click web "password"
    And type web "passwordInput" "password"
    And click web "loginWeb"
    And click web "deviceManagement"
    And click web "searchDeviceWeb"
    And type web "deviceNameWeb" "searchDeviceWeb"
    And the search results should contain "deviceNameWeb"
    And click web "deviceArea"
    Then check device status web
    And open app
    When click app "skipApp"
    And click app "searchBar"
    Then type app "name" "searchBar"
    And the search results should contain "name"
    And click app "reserved"
    And click web "deviceManagement"
    And click web "searchDeviceWeb"
    And type web "deviceNameWeb" "searchDeviceWeb"
    Then check device status web
```

```
    @tag3
    Scenario: Check payment
Given open web
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And open app
When click app "skipApp"
And click app "searchBar"
Then type app "name" "searchBar"
And the search results should contain "name"
And click app "startCharge"
And click web "userManagement"
And click web "searchUser"
And type web "userName" "searchUser"
And click web "user"
Then check user status web

@tag4
    Scenario: Activate user
Given open app
When click app "email"
And type app "emailAdd" "email"
And click app "password"
And type app "passwordInput" "password"
And click app "loginWeb"
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And click web "userManagement"
And click web "searchUser"
And type web "userName" "searchUser"
And click web "user"
Then check user status web
```

```
@tag5
    Scenario: Check charger
Given open web
When click web "email"
And type web "emailAdd" "email"
And click web "password"
And type web "passwordInput" "password"
And click web "loginWeb"
And click web "deviceManagement"
And click web "searchDeviceWeb"
And type web "deviceNameWeb" "searchDeviceWeb"
And the search results should contain "deviceNameWeb"
And click web "deviceArea"
And click web "deleteButton"
And click web "deviceManagement"
And click web "searchDeviceWeb"
And type web "deviceNameWeb" "searchDeviceWeb"
But the search results should contain "deviceNameWeb"
When open app
And click app "skipApp"
And click app "searchBar"
Then type app "name" "searchBar"
But the search results should contain "name"
```

Figure 24: A sliced sample snippet from the test scenarios.

APPENDIX C

USER MANUAL OF THE TEST TOOL

Requirements for Running Test Cases with Cucumber

- To start, we need to have JDK 8 installed on our system.
- We also need the Appium server to be installed on our system.
- We need Android Studio if we're opening our app through an emulator. So, download Android Studio from this official link. Open the exe (Windows) / dmg (Mac) file and follow the instructions to install Android Studio. Alternatively, we can use to connect to a real Android device.
- We need any Java-based IDE (integrated development environment) like Eclipse or IntelliJ. We used Eclipse to write our test cases.

Project Setup in IDE

- Open Eclipse, and then click on File > New > Other.
- Select "Maven Project" and click on the "Next" button.
- Create a simple project" is selected. After that, click on the "Next" button.
- Give the Group ID and Artifact ID, which should be the same.
- Update pom.xml file to add the dependencies for the project Cucumber Java, Cucumber Junit, Selenium Java, Selenium Chrome Driver, Java Client, Cucumber Code in Feature File.
- Create a file called cucumber.feature (name is given as an example). The scenario will define the test. Here, the *Given*, *When*, and *Then* functions define the different features in the test.

Writing Test Cases with Java

- Right-click on src/test/java, and then click on "New."
- After that, select "Class" to create a new class.
- In the next pop-up, give the test file a name. We named it as StepDefiniton.java in our case.
- Implement functions for test cases (You can use Appium Inspector and Selenium IDE to create functions easily).

Automation Testing

To start automated test execution, make sure that the Android device is connected to our system. Run the command adb devices, which will show the connected device. Upload the apk file to emulator or pyhsical device.

Appium Inspector

- Start the Appium server.
- Open Appium Inspector and connect it to the Appium server.
- Select the mobile device or emulator that you want to inspect.
- Save dependencies (See Figure 16)

```
{ "appium:deviceName": "sdk_gphone64_arm64",
  "platformName": "Android",
  "appium:platformVersion": "14",
  "appium:appPackage": "com.google.android.youtube",
  "appium:appActivity": "com.google.android.youtube.HomeActivity"
}
```

- Appium Inspector displays the screen of the mobile device or emulator. You can use Appium Inspector to click on elements, view the locators of elements, and take screenshots. You can also export test code that corresponds to the recorded test session (See Figure 15).

Selenium IDE

- Install Selenium IDE, go to the Chrome web store. `https://chromewebstore.google.com/detail/selenium-ide/mooikfkahbdckldjjndioackbalphokd?pli=1`
- Download the Selenium IDE extension.
- Restart the web browser, and the user will see the Selenium IDE extension.
- Click on "Create a new project" and another pop-up will appear. Mention the name of the project and create new test cases.
- To create a new test case, simply provide the name of the test. The Selenium IDE will redirect to the dashboard where testers can start Record and Playback for the test cases (See Figure 19).
- A button on the top-right corner will take the QA to the target website and start recording the actions or events created while in the record mode (See Figure 20).
- We can export unit cases (e.g., JUnit) that corresponds to the recorded test session (See Figures 21 and 22).

Bridge Pattern Structure

The Bridge Design Pattern [24] is used to minimize the dependency between abstractions and implementations. Essentially, it relies on transforming the relationship between abstraction and implementation from an "is a" relationship to a "has a" relationship with the goal of reducing this dependency. It is one of the structural design patterns, and although it may appear more structurally complex when used, it provides advantages such as achieving a more flexible and maintainable software.

Explanation of the Source Code of the Tool

Drivers

The Drivers interface provides the basis for creating drivers for different platforms. There are two main methods in the interface: readConfig(): This method reads the configuration information required for the device when the application runs. Configuration information is stored in a JSON file. The method is ready to handle IOException and ParseException exceptions. setupDriver(): This method sets up the platform and makes it ready to interact with the device.

DriverAbs

The DriverAbs class implements the Drivers interface and contains two properties named driverConfiguration and driver. This class has two basic methods: readConfig(): This method reads configuration information for the device from a JSON file. The method is ready to handle IOException and ParseException exceptions that may occur during reading. setupDriver(): This method sets up your device and makes it ready to interact with the device. What this method does depends on the type of device.

WebDriver

This WebDrivers class is used to set up a Chrome browser driver and interact with the web application using that driver. The WebDrivers class sets the driver by creating an instance of the ChromeDriver class and assigning it to the driver variable. setupDriver() method initializes the driver, gets the URL of the web application from the Json file. The ChromeDriver class is a web driver provided by the Selenium project. Selenium is an infrastructure for creating automated tests for a web application.

MobileDriver

The MobileDriver class was derived from the DriverAbs class to create drivers for different mobile devices used in the TossI project. The class's setupDriver() method was written to set up an Appium driver for a mobile device and prepare it for interaction. The DesiredCapabilities class is used to store the identification

properties of the mobile device (e.g. device name, UDID, platform name, platform version, application package name, application activity name). The Appium driver was written to establish the device's connection with the Appium server and prepare it for interaction. The MalformedURLException exception can occur when invalid URL is provided and this exception has been prepared to handle. The purpose of this code is to interact with different mobile devices and create mobile device drivers.

AbstractTest

This class is an abstract class called AbstractTest. This class defines the methods to be used by objects derived from the Drivers class. For example, the "runTest()" method is a null method that will be overridden by classes derived from this class. Also, the "clicks()" method is an empty method to click the button given as a parameter. This method will be overridden by classes derived from this class. Finally, the constructor of this class takes object parameter derived from the Drivers class. This constructor assigns the object given as a parameter to the driverObj variable. The purpose of this class is to define methods and variables that are common to classes derived from the AbstractTest class. Users can interact between objects derived from the Drivers class and the Chrome browser driver by using classes derived from this class. For example, the "clicks()" method can be used to interact between the object derived from the Drivers class and the Chrome browser driver. This way, users can run automated tests with the Chrome browser driver.

AppiumTest

AppiumTest class is a class used to test Android and iOS applications through Appium. This class defines methods to be used by objects derived from the MobileDriver class. MobileDriver class is a class used to connect to the Appium server and test Android or iOS applications. This class provides methods for finding and interacting with elements within the application and performing many application testing-related tasks. AppiumTest class is a class that derives from

the AbstractTest class and therefore includes runTest() and clicks() methods. For example, the clicks() method is used to click the button given as a parameter. This method finds the element specified as the button identifier (for example, the ID of the "SkipApp" button) and clicks on it.

SeleniumTest

SeleniumTest class is a class used to test web applications through Selenium. This class contains methods that define various elements (e.g. buttons, input boxes, etc.) in the web application to be tested and perform testing operations for these elements. For example, the clicks method defined in the SeleniumTest class finds the web element that matches the given identifier (for example, the ID of a button) and clicks on this element. This method is designed to work with different browsers supported by Selenium. Additionally, a JSON object named driverConfiguration is defined in the AbstractTest class, which is used to instantiate the SeleniumTest class. This JSON object contains identifying information (e.g. ID, name, etc.) of various elements in the web application to be tested.

StepDefinition

This class is called "StepDefinition" and uses the AppiumTest and SeleniumTest classes. This is a class used with Cucumber to test web applications and mobile applications using Appium and Selenium. Each method defined within the class corresponds to a test step defined by Cucumber to have a specific behavior. For example, the method matching "open web" uses methods in the SeleniumTest class to open the web application. The names of the methods are attempted to match the test steps defined by Cucumber to describe the actions the methods perform. The variables "DriverA" and "WebTest" defined within the class represent the appropriate SeleniumTest or AppiumTest object to use, depending on the type of application we want to test. For example, the method matching the expression "open youtube" creates a new object from the AppiumTest class using the "DriverA" variable, so that tests can be performed using methods in the AppiumTest class.

- Drivers - interface
- DriverAbs — abstract class Driver implement Drivers
- WebDrivers — class extends DriverAbs
- MobileDriver — class extends DriverAbs
- AbstractTest — abstract class
- AppiumTest — extends AbstractTest
- SeleniumTest — extends AbstractTest

REFERENCES

[1] B. Beizer, *Software Testing Techniques*. New York, NY, USA: Van Nostrand Reinhold Co., 2 ed., 1990.

[2] G. Myers, T. Badgett, and C. Sandler, *The Art of Software Testing*. Hoboken, NJ, USA: John Wiley and Sons Inc., 3 ed., 2012.

[3] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 571–579, 2005.

[4] D. Rafi, K. Moses, K. Petersen, and M. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Proceedings of the 7th International Workshop on Automation of Software Test*, pp. 36–42, 2012.

[5] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern Web applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 35–53, 2012.

[6] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Dependency-aware web test generation," in *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification*, pp. 175–185, 2020.

[7] N. Sunman, Y. Soydan, and H. Sözer, "Automated web application testing driven by pre-recorded test cases," *Journal of Systems and Software*, vol. 193, p. 111441, 2022.

[8] P. Kong, L. Li, J. Gao, K. Liu, T. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2019.

[9] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 308–318, 2017.

[10] S. Choudhary, M. Prasad, and A. Orso, "Cross-platform feature matching for web applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, p. 82–92, 2014.

[11] S. Choudhary, "Cross-platform testing and maintenance of web and mobile applications," in *Proceedings of the 36th International Conference on Software Engineering*, p. 642–645, 2014.

[12] S. Choudhary, M. Prasad, and A. Orso, "X-PERT: A Web application testing tool for cross-browser inconsistency detection," in *Proceedings of the International Symposium on Software Testing and Analysis*, p. 417–420, 2014.

[13] R. Lawrence and P. Rayner, *Behavior-Driven Development with Cucumber*. Boston, MA, USA: Addison-Wesley Professional, 2019.

[14] Z. Peng, T.-H. Chen, and J. Yang, "Revisiting test impact analysis in continuous testing from the perspective of code dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 1979–1993, 2022.

[15] M. Wynne and A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[16] S. Hammond and D. Umphress, "Test driven development: The state of the practice," in *Proceedings of the 50th Annual Southeast Regional Conference*, p. 158–163, 2012.

[17] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2 ed., 2004.

[18] M. M. Moe, "Comparative study of test-driven development TDD, behavior-driven development BDD and acceptance test–driven development ATDD," *International Journal of Trend in Scientific Research and Development*, vol. 3, pp. 231–234, 2019.

[19] B. Baker, "Behavior-driven development with cucumber," *Quality Progress*, vol. 53, no. 8, pp. 62–62, 2020.

[20] A. Menegassi and A. Endo, "Automated tests for cross-platform mobile apps in multiple configurations," *IET Software*, vol. 14, no. 1, p. 27–38, 2020.

[21] E. Alegroth, R. Feldt, and L. ryrholm, "Visual GUI testing in practice: challenges, problems and limitations," *Empirical Software Engineering*, vol. 20, pp. 694–744, 2015.

[22] K. Das, *Creating Android, iOS, and Web Drivers on Demand*, pp. 45–60. Berkeley, CA: Apress, 2022.

[23] A. Bruns, A. Kornstadt, and D. Wichmann, "Web Application Tests with Selenium," *IEEE Software*, vol. 26, no. 5, pp. 88–91, 2009.

[24] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[25] L. Bassett, *Introduction to JavaScript Object Notation*. O'Reilly, 2015.

[26] C. Gebizli and H. Sozer, "Impact of education and experience level on the effectiveness of exploratory testing: An industrial case study," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 23–28, 2017.

VITA

Özge AKAT received her Bachelor of Science degree in Computer Engineering from Izmır Institute of Technology in 2020. Since 2018, she has worked at technology startups and companies. She is currently working as an Test Lead & Senior Software Test Engineer for test development. Her research focuses on software test engineering-and test technologies. Besides her professional career, she works as a volunteer in organizations in various positions to simplify animal life.