Remote and parallel test automation at the GUI level using a generic adapter

van der Kuijl, M.G.

Award date:
2021

Remote and parallel test automation at the GUI level using a generic adapter

Master Thesis

Software Engineering

Student name:	M.G. van der Kuijl
Student Number:	
Course:	IM9906 Software Engineering Graduation Assignment
Degree programme:	Open University of the Netherlands
	Faculty of Management, Science and Technology
	Master's Programme in Software Engineering
Supervisor:	Prof. dr. Tanja E.J. Vos
Thesis committee:	Prof. dr. Tanja E.J. Vos, Open Universiteit
	Dr. Pekka Aho, Open Universiteit
	Dr. Machiel van der Bijl, Axini

Contents

Summary

In the field of software engineering, testing at the GUI level is an essential activity that is widely used in industry to perform quality assurance [1]. Besides a manual approach, the automated approach has a variety of available techniques (e.g. capture/replay, model-based, manually scripted) that can be used for effective GUI testing [2, 3]. Performing testing activities in an automated fashion has become the de facto standard nowadays in the software engineering practice [4].

An elegant automated GUI testing technique is implemented in a tool called TESTAR (Test Automation at the user inteRface level) [5]. Using a model-based event-extraction technique, TESTAR is capable of recognizing GUI controls and their properties by interacting with assistive technologies. One example of an assistive technology is the Windows Automation API. TESTAR currently faces several limitations among which include that TESTAR must be installed on the same host of the System Under Test (SUT) when using desktop applications, and test execution can only take place against one SUT instance at a time for both desktop- and web applications. This currently limits the possibility to execute tests within a remote and parallel architecture.

Although it is possible to test web applications using the Windows Automation API, it turned out that this API has a poor recognition of non-native elements in web applications. Changes and updates generated by JavaScript are not always presented to users interacting with the Web through assistive technologies. An effort has been made to embed WebDriver in TESTAR. Although this partly addresses the former of the earlier mentioned limitations, remote and parallel test execution is still a desired functionality.

This research project concerns the development of this desired remote and parallel test execution capability and has resulted in a proposal of a generic API that is capable of integrating multiple backend SUT APIs at the client side using a remote server component. Using an adaptation approach an adapter component wraps around the SUT, manages the low-level details of interacting with the SUT and presents a more abstract view of the SUT to the consumer. An Adapter Framework has been realised to answer several research questions. From the consumer perspective, the communication with the Adapter Frame-

work is done using a RESTful API that provides all necessary capabilities to register, authenticate and setting up a Test between a SUT Consumer and one or more SUT Clients. The interaction between the SUT Client and the server is done using Websocket technology. A proof of concept has showed that it is possible to execute a script-based and scriptless execution approach using the same API. Also, a lot of overlapping and similarities between programmable backend APIs has been observed which makes it theoretically possible to serve multiple Client Adapter types using one front-end API.

Chapter 1

Introduction

In the field of software testing many testing approaches and tools exist [4, 5, 6, 2, 7]. Testing an application via its Graphical User Interface (GUI) is a subject on its own and this research project focuses on testing a system via its GUI using a proposed Adapter Framework. Performing GUI testing activities consists of executing testing activities against a software system with a GUI front-end [8]. Testing a software application at the GUI level is an important step when ensuring software quality of applications with GUIs, mainly because taking the user's perspective is the ultimate way of verifying a program's correct behaviour [9] and ensures realistic tests from the end user perspective [5]. The GUI provides a simple way to control the software system by displaying visual cues such as buttons, text fields, and combo-boxes (also known as "widgets") on the screen. After displaying the widgets users can perform sequences of actions such as mouse clicking or keyboard typing on these visual components [10, 2].

1.1 Automated GUI level testing

To reduce human resources and increase the frequency at which software can be tested, test automation plays an important role [3]. Besides a manual, labour intensive and time consuming approach [10, 11, 12, 13, 14], the more practical automated approaches are used nowadays as an industry practice [15, 16, 17, 5]. Using automated GUI testing approaches, the user's point of view is used while a sequence of available actions against the SUT is automatically executed using tools or frameworks [9].

1.1.1 Defining the GUI

Before going into more details about GUI testing an explanation is needed about how a GUI can be defined formally. The given definition will be used later on in the research project. Memon formally models a GUI as a set of objects/widgets $O = \{o_1, o_2, \ldots, o_m\}$ and a set of object properties $P = \{p_1, p_2, \ldots, p_l\}$ of those objects (e.g., properties like font, caption, etc.) [7]. Mariani et al [18] go a step further by also defining a widget w_i as a pair (type$_i$, P_i) where type$_i$ is the type of the widget and P_i is a set of properties $P_i \subseteq P$.

Each GUI uses certain types of objects with associated properties. At any specific point in time, the state of the GUI can be described in terms of all the objects that it contains, and the values of all their properties. Formally the state of the GUI can be defined as follows:

Definition: The state of a GUI is the set P of all the properties of all the objects O that the GUI contains. A distinguished set of states called its valid initial state set is associated with each GUI.

Definition: A set of states SI is called the valid initial state set for a particular GUI if the GUI may be in any state $S_i \in SI$ when it is first invoked.
The state of a GUI is not static; events performed on the GUI change its state. These states of a GUI are called reachable states. The events are modeled as state transducers.

Definition: The events $E = \{e_1, e_2, \ldots, e_n\}$ associated with a GUI are functions from one state to another state of the GUI. The function notation $S_j = e(S_i)$ is used to denote that state S_j is the state resulting from the execution of event e in state S_i. Events occur as part of a sequence of events.
The possible sequences of available actions can be manually defined and listed using script-based approaches or automatically determined using scriptless approaches. The possible sequences of available actions also known as a valid event sequence are sequences that are permitted by the structure of the GUI and following Memom, can be defined as follows:
Definition: A legal event sequence of a GUI is $e_1; e_2; e_3; ...e_n$; where $e_i + 1$ can be performed immediately after e_i

From this definition and the definition given in paragraph 1.1.1 the following definition applies for a Test Case:

Definition:. A GUI Test Case T is a pair $\{S_0, e_1; e_2; ...; e_n\}$, consisting of a state $S_0 \in S_I$, called the initial state for T, and a legal event sequence $e_1; e_2; ...; e_n$.

Testing at the GUI level can be viewed from various perspectives. The way how a GUI Test Case is created and how it gets executed against the SUT once it is created. Both are discussed in the next sections.

1.1.2 Test Case Design

Before automated GUI level testing can take place, first a legal event sequence of the SUTs GUI must be created as part of the Test Case T. This section discusses several techniques that can be used to design a Test Case.

Manually using Capture-replay tools
A step towards GUI Test Case design is the usage of capture-replay tools. Capture-replay tools facilitate the tester generating test scripts and automated execution of these scripts against the SUT [10] [12]. Before the test script is generated the capture component of the tool stores the end-user interaction actions in a script file that can be replayed later using the replay component of the tool. While capture-replay tools partially address the problems of testing GUIs, they have drawbacks of their own, including problems involving test script maintenance once the GUI starts to evolve [19] and the support for automatic Test Case generation [12].

Model based testing (MBT)
Another step further than using Capture-replay tools is the automatically generation and execution of Test Cases. This can be achieved by performing a model-based testing approach. The goal of model-based testing is to check the conformity between the implementation and the specification (model) of a software system [12]. One benefit of a model-based testing approach is Test Case generation (from the specification), Test Case execution and the comparison of the actual results obtained from the implementation with the expected results described by the specification [10].

Random testing
Besides the manual Test Case design and the MBT approach, a random testing approach exists. Using a random testing approach the GUI of the SUT is walked randomly and all encountered events are executed in sequence by first extracting the possible events followed by action selection, creating the Test Case and executing the Test Case as an event sequence [2] against the SUT. This event-extraction can be accomplished by interacting with the GUI via accessibility APIs that are provided by development platforms and used at runtime [10].

Model-based testing (MBT) approaches for GUI-driven software construct an abstraction of some subset of an application and prescribe a Test Case selection process which constructs Test Cases based on the model. Model-based GUI testing approaches first generate testscripts ”offline” before executing them against the SUT [10]

1.1.3 Test Case Execution

Test Case execution is briefly discussed in this section. Once a Test Case T is designed using one of earlier discussed approaches the Test Case can be executed against the SUT using several ways. Besides a manual Test Case execution approach, both script-based and scripless Test Case execution will be discussed.

Manual Test Case execution
As discussed earlier, the most basic testing approach is manually executing the

Test Case against the SUT. Using this approach, a valid sequence of actions is defined first and eventually manually executed against the SUT by the tester of the SUT.

Script-based Test Case execution
GUI testing effort can significantly be reduced using a script based testing approach. Once the desired sequence of events is defined by the tester and combined in a Test Case, it may be reused to automate GUI interactions in regression testing [2]. This testing approach gives several challenges amongst the one that can be seen by using a capture-replay approach discussed later. Because script writing is a labor-intensive process, compared with Model-based testing, also discussed later, the number of Test Cases is often small and the maintenance of test scripts once the software starts to evolve is a challenge [20]. After all, a Test Case can break once the GUI starts to change and a once valid event sequence is invalidated by the change.

Scriptless Test Case execution
When scriptless testing is performed, no offline Test Cases are crafted upfront and executed against the SUT at a later moment. Several techniques and tools exist to enable this testing approach. Without a script it is not possible to define a sequence of events and combine these events in a Test Case. Therefore, another way of defining a Test Case should be used. Test Cases can be generated automatically by interacting with the backend APIs that exposes the possible events E. To generate Test Cases, we can, for instance, start from a known initial state $S0$ and use a graph traversal algorithm, enumerating the nodes during the traversal, on the event-flow graphs [21].

1.2 Localization strategies

To actually determine what sequences of available actions are available, several localization strategies exists [4, 20]: coordinates-based, widget-based and visual-based. These strategies are based on techniques used to stimulate and assert the SUT. In this work, we will only concentrate on the widget-based approach discussed briefly below.

The widget-based localization strategy
The widget-based approach is a suitable technique that can be used for applications exposing their GUI as hierarchical data structures and will be used within the context of this research assignment. Using this technique, applications locate elements using information contained in the widget-tree. As defined earlier in Section 1.1.1, this tree provides a data structure with a fixed set of properties, and the complete data structure with the set of properties at a given time constitutes the state of the GUI [22]. In case of a web application elements, like anchors and buttons, the widgets are located by accessing their properties (eg, identifier or text) or by navigating the DOM using Xpath queries [23]. The

actual interaction is accomplished by interacting with APIs like WebDriver. In case of desktop applications widgets, like buttons and menu items, the widgets can be accessed by interacting with APIs like the UI Automation API.

1.3 GUI testing using TESTAR

A Widget-based random testing approach is implemented in a tool called TESTAR [5]. Using a GUI-level dynamic event-extraction technique against the Accessibility API of the host operating system, to communicate with desktop applications, and the Java access bridge to communicate with Java Swing applications. TESTAR is capable of recognizing GUI controls and their properties by interacting with these APIs using the system mouse and keyboard. Using this approach, TESTAR indirectly performs programmatic interaction with the SUT. A powerful core capability of TESTAR is selecting actions based on information derived from the GUI and the creation of test sequences on the fly [24]. This in contrast to most model based techniques that first create Test Cases offline and execute them later as scripts against the SUT [10].

In a typical setup, TESTAR runs on the same machine as the SUT. After TESTAR is started, the SUT is started and TESTAR scans the SUT to retrieve the GUI state as a Widget Tree. The Widget Tree is translated into a set of sensible actions that a user could execute in a specific SUT's state and one action is selected and executed against the SUT. A new Widget Tree is retrieved, and available Oracles are applied to check (in)validness of the new UI state.

Chapter 2

Related work

This chapter discusses the usage of programmable APIs that can be used by SUT consumers. First several Accessibility APIs are described and explained and from there the concept of programmable APIs are explained.

2.1 Accessibility APIs

An accessibility API is an application programming interface (API) by which an application (server) exposes its graphical user interface (UI) and content to another application (the client). The Accessibility API is a fundamental component of digital accessibility. Through the accessibility API, the client discovers, represents, and modifies the server's UI and content. An example of accessibility servers can be any user agent, like a web browser or document viewer. Accessibility clients can be assistive technologies, like screen readers or magnifiers, UI automation and testing scripts, or dynamic content scripts like JavaScript scripts [25]. Widgets that are visually displayed on the screen have many properties like size and physical location in the viewport. Besides the visible properties, there are many invisible properties that are present for each object and not related to visual appearance at all. These properties uniquely identify an object and give information about the type of object (checkbox or button) and the state (checked or enabled). Web frameworks, operating systems and other platforms provide a set of interfaces that expose information about these objects and events to assistive technologies.

Microsoft UI Automation

The Windows Operating system uses the Microsoft UI Automation accessibility framework [19]. This framework, a Windows implementation of the UI Automation API enables Windows applications to provide and consume programmatic information about user interfaces. Together with the previous Microsoft Active Accessibility the ecosystem of Windows automation technologies, the Windows Automation API is formed [26].

The Windows Automation API exposes every piece of the UI to client applications as an automation element where the Providers supply property values for each element. All Elements are exposed as a tree structure, with the desktop as the root element. Automation elements expose common properties of the UI elements they represent. One of these properties is the control type, which describes its basic appearance and functionality (for example, a button or a check box) [26].

Accessible Rich Internet Applications and HTML Accessibility API Mappings

The Accessible Rich Internet Applications Suite (WAI-ARIA), defines a way to make Web content and Web applications more accessible to people with disabilities. It especially helps with dynamic content and advanced user interface controls developed with HTML, JavaScript, and related technologies [27]. By supplementing HTML with specific attributes these can be based to assistive technologies when there is not otherwise a mechanism. For example, ARIA enables accessible navigation landmarks in HTML4, JavaScript widgets, form hints and error messages, live content updates, and more. Many ARIA widgets are currently incorporated into HTML5. Part of the WAI-ARIA is the HTML Accessibility API Mappings (HTML-AAM). HTML-AAM defines how user agents map HTML [HTML] elements and attributes to platform accessibility application programming interfaces (APIs) [28].

2.2 Programmable Web APIs

This section discusses programmable APIs that can be used to interact with Web applications. First, the Document Object Model is discussed briefly, after which the WebDriver specification and several implementing frameworks are discussed, in addition to the DevTools protocol implemented by the Puppeteer NodeJS library.

The Document Object Model

For a web application, the structure of a web page that is part of the application is represented as a Document Object Model (DOM). The DOM is an API for accessing and manipulating documents (in particular, HTML and XML documents) [29]. The Document Object Model (DOM) is a cross-platform and language-independent API that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document. The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree. These methods can change the structure, style or content of a document. Nodes can have event handlers attached to them. Once an event is triggered, the event handlers get executed. Conforming the definition given in 1.1.1, all elements of the DOM at a certain point in time can be seen as the state of the application at that time.

The WebDriver specification

For modern web applications, the WebDriver specification can be used to determine what exact sequence of events can be executed against a Web Application Under Test (WAUT). The WebDriver specification provides a well-designed object-oriented API that provides improved support for modern advanced web-applications created through dynamic web pages. The WebDriver specification provides an API to access elements on a Web page as they are rendered in a browser [30]. The specification leans on the DOM, UI Events and the ECMAScript Language Specifications. Since June 2018, the WebDriver specification is recommended by the W3C with the goal to draw attention to the specification and to promote its widespread deployment.

The WebDriver API consists of the WebDriver interface with a concrete implementation done in two classes: RemoteWebDriver and HtmlUnitDriver. For every Browser type or App operating system a subclass is extended from the RemoteWebDriver class.

The RemoteWebDriver acts as a middle man like browser-driver (eg: chrome-driver) behaving like a server that sits between the automation script and the browser. This enables the browser control.

To facilitate developers and testers performing test activities against the WAUT, the WebDriver specification is implemented in several industry frameworks and tools like Appium, Selenium WebDriver and WebdriverIO.

Appium
Appium is an industry tool that is frequently used to support automated approach against native, hybrid and mobile web apps. Appium is an open source test automation framework that drives several operating systems, like iOS, Android, and Windows apps using the WebDriver protocol [16]. Appium uses a script-based testing approach against its backends using WebDriver libraries and APIs. Appium offers an HTTP server written in Node.js, a JavaScript runtime built on Chrome's V8 JavaScript engine that creates and handles WebDriver sessions [31].

Selenium WebDriver
Selenium Webdriver is an open-source collection of APIs used for testing web applications. The Selenium Webdriver tool is used for automated testing of web applications to verify whether they work as expected or not. It supports all common browsers browsers like Firefox, Chrome, Safari, Internet Explorer and Edge. It is also capable of executing cross-browser testing.

The following testscript written in the javascript based language Node.js [30] makes use of the selenium-webdriver Node.js library. Combined with the Firefox driver, once this script is executed, it first navigates to google.com, finds an element with name 'q', types Selenium in this element and presses the Enter key:

```
const {Builder,Buy,Key,util} = require("selenium-
    webdriver");

async function example()

{

let driver = await new Builder.forBrowser("firefox").
    build();

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

await driver.findElement(By.name("q").sendKeys("
    Selenium",Key.RETURN));

}

example();
```

WebdriverIO
Like Appium, WebdriverIO uses a script based testing approach using the WebDriver specification to integrate with the WAUT. WebdriverIO implements the WebDriver specification using Node.js and allows to run tests on desktop and mobile applications [17].

Puppeteer and the DevTools protocol

Like Webdriver, Puppeteer provides a high-level API to control a Web Application. Most things that can be done manually in the browser, can be done using the Puppeteer API. A capability that is important for the context of this research project is UI testing. The Puppeteer API is hierarchical and mirrors the browser structure. Communication with the browser is made possible using the DevTools Protocol. All communication is made possible by using serialized JSON objects of a fixed structure [32].

DevTools Protocol
The Developer Tools Protocol is used by various Browsers, JavaScript Engines and debugging tools. The Chrome DevTools Protocol allows for tools to instrument, inspect, debug and profile Chromium, Chrome and other Blink-based browsers. Many existing projects currently use the protocol. The Chrome DevTools uses this protocol and the team maintains its API.

2.3 Programmable Desktop APIs

For applications running as desktop applications, programmable APIs are available to automate the testing of Windows Desktop Applications. The researched UI Automation API discussed next is an implementation of an Accessibility API discussed in section 2.1. According to [25] and observing the industry, there are several accessibility APIs for different operating environments available. Concerning the scope of this research, only the Windows UI Automation API will be briefly discussed in this section.

UI Automation Specification
The UI Automation specification forms the basis of the Windows implementation of UI Automation. The usage of UI Automation is twofold. First by using UI Automation and following accessible design practices, developers can make applications running on Windows more accessible to many people with vision, hearing, or motion disabilities. Secondly, UI Automation is specifically designed to provide robust functionality for automated testing scenarios. The latter is the subject of this research, where TESTAR already consumes this API and implements functionality for automated testing using a scriptless approach.

The Windows OS - UI Automation API
The Windows Operating system offers two APIs that can be used for UI accessibility and software test automation. The nowadays available Automation API has followed a long evolution path which has led to the offering of two API specifications. As a platform add-on the legacy Microsoft Active Accessibility was introduced in 1996 to Windows 95. The second API is an implementation of the User Interface Automation specification (UI Automation) and is introduced in Windows Vista and version 3.0 of the .NET Framework. The UI Automation Specification can also be supported across platforms other than Microsoft Windows.

Windows UI Automation tree overview
The UI Automation tree is a structure where the root element represents the Windows desktop window (”the desktop”) and whose child elements represent application windows. Each of these child elements can contain elements that represent pieces of the UI. These elements are represented as Microsoft UI Automation control types.

Control types are properties that serve as well-known identifiers that indicate the kind of control that a particular UI element represents. Control types are exposed as UIALocalizedControlType objects in the form of Button Control type or a Menu Control Type for a Button or a Menu. A Microsoft UI Automation Client application can use this type to identify the capabilities of a control and to determine how to interact with it.

Views of the UI Automation tree
Looking at the UI Automation tree, several views can be distinguished, each providing relevant information provided to clients. The Raw View gives the most detailed information available and consists of the full tree of automation elements with the desktop as the root element. This view closely follows the native programmatic structure of an application. A subset of the raw view can be found in the control view. This view only includes items that satisfied the control element property. Items holding this property provide information to the user or enable the user to perform an action. These are the UI items that are most interesting to automated testing applications.

Automatically testing an application using an API means that a sequence of events first gets invoked against an intermediate component e.g. the Webdriver API for web applications or the Windows Automation API for desktop applications. This intermediate layer translates this sequence of events to instructions that the SUT understands. On behalf of the initial caller this application layer invokes the SUT, asserts and redirects the output back to the caller.
Automated testing of Web application can be achieved successfully using the currently W3C draft version of WebDriver API. WebDriver is a remote control interface that enables introspection and control of user agents. WebDriver provides a platform- and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers [17]. The WebDriver specification is a W3C recommendation which means this specification can be considered as a Web standard. According to the specification, Webdriver is used to write tests that automate a user agent, like a web browser, from a separate controlling process. The WebDriver API provides a set of interfaces to discover and manipulate DOM elements in web documents and to control the behavior of a user agent. WebDriver supports both script-based and scriptless approaches.

Automated testing of web applications

Testing effort is often a major cost factor during software development. Many software organizations are spending up to 40% of their resources on testing [4]. Therefore, an existing open problem is how to reduce testing effort without affecting the quality level of the final software. Automation is a major solution for reducing high testing effort. Automating certain manual tasks from software testing process can save a lot of testing time. It can help in performing repetitive tasks more quickly than manual testing. Software testing can be divided into two main categories, manual testing and automated software testing. Both categories have their individual strengths and weaknesses. When testing web applications in an automated approach, actions are invoked using a script-based or scriptless test case execution.

Using the script-based approach, the script is crafted upfront by the test automation engineer. A test script contains a sequence of commands or events, combined with test oracles that assert the outcome of a test step, stored in a script language file to execute a test case and report the results. The script may contain logical decisions that affect the execution of this script, creating multiple possible pathways, constant values, variables whose values change during playback. The advantage of test script development process is that scripts can repeat the same instruction many times in loops, each time with different data. Once the desired sequence of events is defined and combined in a Test Case, it can be executed against the SUT.

When a scriptless testing approach is performed, no offline Test Cases are crafted upfront and executed against the SUT at a later moment but the test case generation and execution are combined. Several techniques and tools exist to enable this testing approach. Without a script, it is not possible to define a sequence of events and combine these events in a Test Case. Therefore, another way of defining a Test Case should be used. Test Cases can be generated automatically by interacting with the backend APIs that exposes the possible events E. To generate Test Cases, we can, for instance, start from a known initial state S0 and use a graph traversal algorithm, enumerating the nodes during the traversal, on the event-flow graphs [21].

Chapter 3

Research Questions

The introduction shows there are many ways to test an application via it's graphical user interface. One way is to use a Widget-based random testing approach at the GUI level. Using this approach, scripted and a scriptless testing activities against a SUT can be achieved. An environment specific API that is currently available for desktop applications is offered by the Microsoft UI Automation accessibility framework. This API is also used by TESTAR to implement its automated scriptless testing approach against desktop SUTs. For Web based applications, several standards are available, for example, WebDriver and Puppeteer both are capable of inspecting and executing actions against Web based SUTs.

This research project will first bring the Widget-based random testing approach at the GUI level together with testing Web-, Native- and Desktop Application using a single API. Secondly an Adapter Framework will be created that implements this API. While designing and defining the API and experimenting with an initial implementation, the research questions summarized in the following subsections will be answered.

Research questions
The main research question asked in this research project is:

> How can a **generic API** be implemented for **remote and parallel** GUI testing and **validated for different type of consumers**?

To answer the main research question, the following sub research questions must be answered:

RQ1: What does it mean for an API to be generic and how to adapt to multiple environment specific APIs like the Windows Automation API, the WebDriver API and the Android API.

RQ1.1: What is the overlap between the Windows Accessibility API and other specific backend APIs like the Webdriver API and/or the Android API;

RQ1.2: Are there any generic APIs available yet and what properties hold for these APIs; (literature study)

RQ1.3: How would it be possible to translate to a more generic version of the Windows Automation API, the Webdriver API and the Android API.

RQ2: What are the challenges and possible solutions looking at remote and parallel model-based GUI testing.

RQ2.1: What protocols, tools and frameworks can be used to support remote and parallel execution of DOM based event-extraction testing approaches;

RQ2.2: What design- and/or anti patterns are available?

RQ2.3: What non-functional aspects are important to execute remote and parallel model-based testing.

RQ3: How can the implementation be validated by several consumers?

Validating the implementation will be done by integrating a script-based consumer (e.g. WebDriver) and a scriptless consumer (e.g. TESTAR).

Assignment part one - The Generic API
The first part of the research will be entirely devoted to defining and designing a generic API that is capable of driving GUIs, as formally defined in Section 1.1.1. The main goal is modeling an API that abstracts away environment specific APIs to the generic API. Research question RQ1 and its sub questions will be answered in the first part. To define the generic API the formal definition of the GUI (given in paragraph 1.1.1) will be used and combined with both the WebDriver specification and the API specification provided by the Windows automation technologies ecosystem. With the current knowledge, the research assignment will kickoff with the following to be determined during the research:

- the possible types of objects/widgets that are part of O
- how the possible types of objects should be represented in a generic way
- the possible types of properties P
- how the possible types of properties should be represented in a generic way
- how a state can look like and can be represented

The API specification of the generic API will be made available as a Java API for both the client and server part. JSON will be used for the communication between the SUT clients and server. A JSON API specification will be made available as a Web Application Description Language (WADL) definition [33] combined with a JSON Schema definition [34].

Assignment part two - The Adapter framework
The second part of this research builds further on the first part and addresses the desired functionality to support remote test execution. An adapter framework (client and server part) is implemented with at least the following core capabilities:

- translating from backend SUT to abstract view and vice versa
- managing the low-level details of interacting with the SUTs.
- providing the possibility to remotely communicate between multiple SUT clients and a central server
- facilitating all low-level communication between the SUT clients and the server

For the adapter framework the following design choices have already been made in the research proposal and will be discussed further:

- the communication between the SUT clients and server will be based on WebSockets combined with NodeJS
- JSON will be used as the data format
- the adapter framework will support both local and remote test execution

The design choices are based on a combination of upfront requirements given by the research committee (the usage of Websockets) and best practices and standards used in the industry (usage of JSON and NodeJS) as described in the research context. Gaining knowledge about the first will be in scope of this research assignment, basic knowledge about the latter is already gained by following several courses about the topics. Further hands-on experience and knowledge will be gained during this research.

To validate the results obtained from researching both parts, an adapter framework will be developed, and a proposal, together with an initial implementation, will be made to integrate TESTAR with this adapter. This will validate a scriptless integration. A script-based integration will be validated by reusing and modifying yet to be determined test scripts that are used, for example, by Selenium WebDriver or the WebDriver.io framework.

Chapter 4

Generic API

The goal of this chapter is elaborating the first research question (RQ1) by trying to understand what it exactly means for an API to be generic and how an adaption to multiple environment specific APIs can be made possible.

4.1 Characteristics of the Generic API

An important design objective that the API fulfills is exposing a framework, independent of back-end programming languages and operating systems. This framework is capable of providing all the desired functionality to consumers of this API. Through the use of this API, the underlying implementation is hidden and an uniform language and data structure is used. In the context of this research, this objective has led to two technological choices. First, the usage of the REST software architecture style [35], and secondly, a formally defined common information model that is capable of testing multiple backend SUTs at the GUI level using a script-based and a scriptless testing approach. Both choices are made in order to keep the development within the earlier mentioned design objective. Both choices are discussed in detail in the following subsections. Besides the usage of JSON over HTTP at the consumer side and Websockets at the SUT client side, a formally defined common information model is defined to ensure loose coupling between SUT Consumers and SUT Clients.

Translating this objective to the scope of this research project, this API should be able to present a data structure that is capable of expressing the following:

The set of Objects/Widgets

To be able to present a complete set of Objects/Widgets this set should be listed for all possible SUT Clients (Web, Desktop, Native App). With the following definition as a starting point :

O **is a set of objects/widgets defined as** $O = \{o_1, o_2, ..., o_m\}$**.**

For a Web based application all objects are expressed as HTML tags and can be listed as the following set:

$$O = \{< a/ >, < li/ >, < div/ >, < img/ >, < h1/ >, < small/ >, < ul/ >, ...\}$$

For a Windows Desktop application with the use of the Automation API, all objects are expressed as AutomationElements. A subset can be expressed as following:

$$O = \{UIAFullDescription, UIAIsTopmostWindow, UIAIsDialog, UIALocalizedControlType, ...\}$$

The set of Objects/Widgets properties

All the objects in O contain properties. The set of properties can be defined as following:

P be a set of object properties $P = \{p_1, p_2, ..., p_l\}$

These properties give information about the object itself like it's unique id or name. For the purpose of this research, it is also important to know if an object is actionable. For an Object to be actionable, it must contain at least one property that expresses it is actionable.

For a Web based application, an object is actionable if this object contains specific HTML Event Attributes, like Keyboard-, Mouse-, Drag or Clipboard Events:

$$P = \{onClick, onMouseUp, onScrollDown, ...\}.$$

The following listing shows a HTML Button with an onclick attribute. This Button invokes a JavaScript function once it is clicked giving this Button the actionable characteristic:

```html
<button onclick="myFunction()">Click me</button>
```

Listing 4.1: an actionable HTML object

For a Windows Desktop application and from the perspective of the UI Automation API, some properties are common to all Automation Elements. For instance, all Elements possess a name or textual description, and a role (UIAButton) or type (UIALocalizedControlType). Other attributes, like value and state, vary in applicability to different types of Automation Elements. The role of an Automation Elements is the attribute that best summarizes the nature and functionality of the Element. Examples of types are “button”, “menu item”, “list” or “document”. Knowing the type of an object, a SUT consumer may determine what other properties are relevant in order to represent the object, and what general behavior the Element may exhibit. For example, a "button" is described by its name property and pressed state, and it can be used to invoke an action when the UIAIsInvokePatternAvailable is true. In the UI Automation API, in addition to a single type property, the nature and functionality of an AutomationElement is specified by a combination of predefined behavior or pattern. This approach allows a more flexible, accurate description of the object. An AutomationElement, for instance, is actionable if this object contains the **UIABoundingRectangle** property or the **UIAIsInvokePatternAvailable** property. The former indicates where the AutomationElement is exactly located on the screen, and the latter indicates if it can be invoked using the Invoke pattern, if the value is true for this property. For the **UIABoundingRectangle**, a mouse click in the middle of this rectangle can be invoked (assuming the button is not obscured when a mouse click is simulated). The possible subset of properties for a desktop application looks as following:

$P = \{UIAIsInvokePatternAvailable, UIABoundingRectangle, ...\}$

4.2 Analysis of the overlap between backend APIs

In section 2, some overlap can be discovered between programmable APIs. Based on this, a further analysis is performed to determine how an API could possibly look like if it must serve multiple backend APIs.

The following categories are distinguished to perform a structured analysis:

- **category 1: extracting the state tree**
  an understanding must be available about the information that is needed to extract the state of the GUI of the SUT for all SUT types.
- **category 2: locating elements**
  it should be clear what information is needed to locate an element on the GUI of the SUT for all SUT types.
- **category 3: executing a sequence of actions**
  because every programmable API has it's own way to communicate with the GUI of a SUT, this category focuses on the way actions can be executed against actionable items.

Category 1: Extracting the state tree

Once a SUT ends up in the state and exposes its state tree, this state tree must be composed and returned to the SUT consumer. For every type of SUT, another implementation of a tree traversal algorithm can be used. For the Web based and Windows Desktop based SUTs, this section describes the information that is needed to be able to start composing the state tree of the SUT.

Extracting the state of a Web API
For a Web-based SUT, the state tree is expressed as the set of HTML tags with their properties that are part of the Document Object Model of the web page that is fully loaded. A tree traversal of this DOM can be executed using Web scraping algorithms using the APIs provided by e.g. WebdriverIO or Puppeteer. Information like the destination (URL) to navigate to, before the state tree can be extracted, must be available first. Once the backend API successfully navigated to this URL, some time might be needed for the web page to fully load.

The DOM tree can be imagined as a collection of "nodes" related to each other through parent-child and sibling-sibling relationships. Each node represents an object in an XML document, including elements, textual content, and comments. Each XML document contains a single root element (¡html¿ in HTML, for example) from which all other nodes ultimately descend.

DOM tree traversal may be accomplished through the use of six basic properties **previousSibling**, **nextSibling**, **childNodes**, **firstChild**, **lastChild** and **parentNode**. All properties, except childNodes, contain a reference to another node object. The childNodes property contains a reference to an array of nodes.

Extracting the state using a Desktop API
For the Windows Desktop Application, the Windows Automation API can be used to extract the state tree of a Desktop application. If the Automation API is used, the consumer of this API acts as the UI automation client. UI Automation clients view the UI Automation elements on the desktop as a set of AutomationElement objects arranged in a tree structure. For a Windows Desktop SUT, the state tree can be extracted by first looking for the process id belonging to the name of the application that acts as the SUT.

Using the TreeWalker class, a tree traversal algorithm can be implemented. A client application can navigate the UI Automation tree by selecting a view of the tree and stepping from one AutomationElement to another in a specified direction using the **GetFirstChild**, **GetLastChild**, **GetPreviousSibling**, **GetNextSibling**, and **GetParent** methods.

Category 2: Locating elements
An important part of the state tree is the set of items that are actionable. This information is needed by the SUT Consumers to know that one or more actions can be invoked against these items. Based on the definition for the set of objects from section 4.1 and the definition of the set of actionable properties P from section 4.1 for every item in O the information requirement can be analysed further.

Locating elements using Web APIs

For the Web based API, there are several Locator strategies to find elements. Using the widget based localization strategy as described in section 1.2, the id of an element or the location path expressed as an XPath expression is the least information that should be available in the state tree. If the Web application uses CSS to style the web pages, also a CSS selector can be used. Using an XPath locator strategy, in Puppeteer also called ElementHandle that represents an in-page DOM element can be found as following:

```
const puppeteer = require('puppeteer');

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  const element = await page.$('<xPath>');
  // ...
})();
```

Listing 4.2: Puppeteer API locating element example

Locating elements using Desktop APIs

To locate elements that are part of an Windows Desktop application, the Automation API offers various ways to obtain AutomationElement objects for user interface (UI) elements. A best practice for finding UI Automation elements in automated testing scenarios is by its name, AutomationId or some other property or combination of properties. The FindFirst method is the easiest to use method. If the element sought is an application window, the starting-point of the search can be the RootElement.

Initially and after performing a sequence of actions, a full state tree is needed for a SUT consumer. In this case, constructing a subtree of all elements of interest can be composed by using the TreeWalker class. Using the Automation API, the same is accomplished by starting from the root AutomationElement, lookup the desired application window, and from there, lookup the desired Automation Element.

```
var root = AutomationElement.RootElement;
AutomationElement element = treeWalker.GetFirstChild(root);

// Look for the window named "Application Window"
var applicationWindow = new PropertyCondition(
    AutomationElement.NameProperty, "<Application Window>")
    ;

var treeWalker = new TreeWalker(applicationWindow);
AutomationElement element = treeWalker.GetFirstChild(root);

// now look for a button with the text "Button Name"
var buttonElement =window.FindFirst(TreeScope.Children, new
     PropertyCondition(AutomationElement.NameProperty, "<
    Button Name>"));
```

Listing 4.3: Automation API locating element example

To achieve this, an element that holds the property UIAIsInvokePatternAvailable with the value 'true' can be invoked using an IUIAutomationInvokePattern interface from the element to invoke the action of a control.

Category 3: Executing a sequence of actions

Once items are identified and the state tree is extracted from the SUT, the next step is executing one or more actions against a given widget. Not only the requirement that an item is actionable plays a role now, but also what kind of item we are dealing with. A button, for instance, can only be pressed and in a textbox an input sequence of characters can be invoked using the keyboard action. The opposite is not possible.

Executing actions using Web APIs

For a Web Application, performing actions against an element once this element is located can be achieved as following:

```javascript
const puppeteer = require('puppeteer');

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  const element = await page.$('<xPath>');
  await  element.click ();
  // ...
})();
```

Listing 4.4: Puppeteer API locating element example

Executing actions using Desktop APIs

Using the Automation API, the same can be achieved, assuming the InvokePattern is available for a given Automation Element:

```csharp
// click the button using the InvokePattern
var invokePattern = buttonElement.GetCurrentPattern(
    InvokePattern.Pattern) as InvokePattern;

invokePattern.Invoke();
```

Listing 4.5: Automation API locating element example

Analysis of the results

Based on the information analysis done in this section, a proposal can be made towards an API that is capable of driving multiple programmable APIs. For every category, the following attributes are needed:

Category 1: extracting the state tree

- ***For Web Applications***
  - (mandatory) destination (URL)
  - (optionally) how long to wait before extraction must start
  - (optionally) from what element node of the DOM tree to start
- ***For Desktop Applications***
  - (mandatory) name of the application window

Category 2: locating elements

- ***For Web Applications***
  - (choice)
    - CSS Selector
    - XPath Selector
    - Tag name
    - Link text selector
    - Partial Link text selector
- ***For Desktop Applications***
  - (choice)
    - XPath Selector
    - AutomationId

Category 3: executing a sequence of actions

- ***Type of Widget***

4.3 Driving the SUT

To change the state of a SUT, one action or a sequence of actions should be executed against the SUT. Human Interface Devices (HID), like a keyboard, a mouse, or a touchscreen, are used to perform these actions. One action can be divided into a main action, like a keyboard or a mouse action, and one or more sub actions, like pressing a key using the keyboard main action, or moving a pointer using the mouse main action. The sub action is always modelled as a 2-tuple name/options. The options is a set of key-value pairs (see Figure). E.g. the keyboard main action contains a sub action with name ”type” and options with a key text and the value that must by typed.

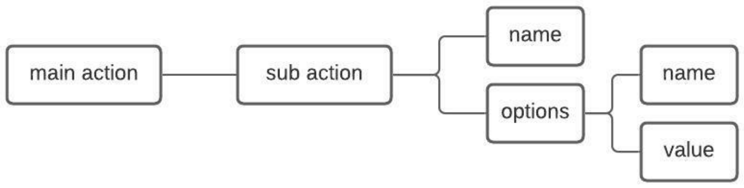

Figure 4.1: Action sequence structure

Keyboard sub actions

For the keyboard main actions, the following sub actions are available to interact against the SUT using a keyboard:

- **type**, sends a key down, key press/input, and key up event for each character in the text.
- **press**, if the key is a single character and no modifier keys besides the indication that the Shift key is being held down, a key press/input event will be generated. The text option can be specified to force an input event to be generated.
- **down**, dispatches a key down event. If the key is a single character and no modifier keys besides Shift are being held down, a key press/input event will also be generated. The text option can be specified to force an input event to be generated. If the key is a modifier key, Shift, Meta, Control, or Alt, subsequent key presses will be sent with that modifier active. To release the modifier key, use the up sub action.
- **up**, Dispatches a key up event.
- **sendCharacter**, dispatches a key press and input event. This does not send a key down or key up event.

Mouse sub actions

The following mouse sub actions are available to interact with the SUT using a mouse:

- **down**, dispatches a mouse down event.
- **move**, dispatches a mouse move event.
- **up**, dispatches a mouse up event.
- **click**, shortcut for mouse.move, mouse.down and mouse.up.
- **wheel**, dispatches a mouse wheel event.

Starting and stopping the SUT

Starting and stopping the SUT depends on the type of SUT. For a web based SUT client, for instance, starting the SUT depends on the fact that the web application is up and running and accepting requests. For web applications, it can be assumed that this SUT is always running as a web application in a web server container. For specific clients, only the SUT client adapter knows how to start the SUT. This means a generic action is not applicable. Nevertheless, the adapter framework exposes a management API that is able to check the status of the SUT and sending specific commands to the SUT to start, shutdown or restart the SUT or shut it down via the adapter framework.

Navigating through a SUT

Navigating through the SUT is possible by round tripping between SUT Consumer and SUT Client by executing one TestStep. Every round trip starts with optionally executing a sequence of actions, followed by extracting the state of the SUT. The Adapter Framework translates the query and invokes the SUT via the SUT specific SUT Client Adapter. The SUT Client Adapter populates the current WidgetTree and gives this WidgetTree to the adapter framework. The SUT Consumer determines what action can be performed by selecting an action from the set of O and P. Once this action is selected, it should be translated to a corresponding Keyboard or Mouse action. After translating to the corresponding action, the exposed REST API of the adapter framework can be invoked. This time, a request with actions is sent, these actions are executed against the SUT that should change the state of the SUT. The new state is extracted again and returned to the SUT consumer.

Extract the current state of the SUT

Extracting the current state of the SUT is an implicit action that can be executed in two ways. First, when an empty sequence of actions is sent to the Adapter Framework, only the state tree of the SUT is extracted and returned to the consumer. Secondly, in case of sending actions, these actions are executed before the new state is extracted and returned to the consumer. The API calls are discussed in section Generic API.

Chapter 5

The Adapter Framework

The Adapter Framework that is build as part of this master thesis is discussed in this section. With this Framework, a proof of concept is created and elaborated. This Adapter Framework implements the proposed API discussed in section 4. The goal is to answer the second research question (RQ2) and its sub questions.

High level overview, global functionality and design choices
The Adapter Framework can be broken down into several parts, all working close together to meet the overall requirements looking at remote GUI testing against a generic API. Figure High level overview, global functionality and design choices gives an overview of the created framework.

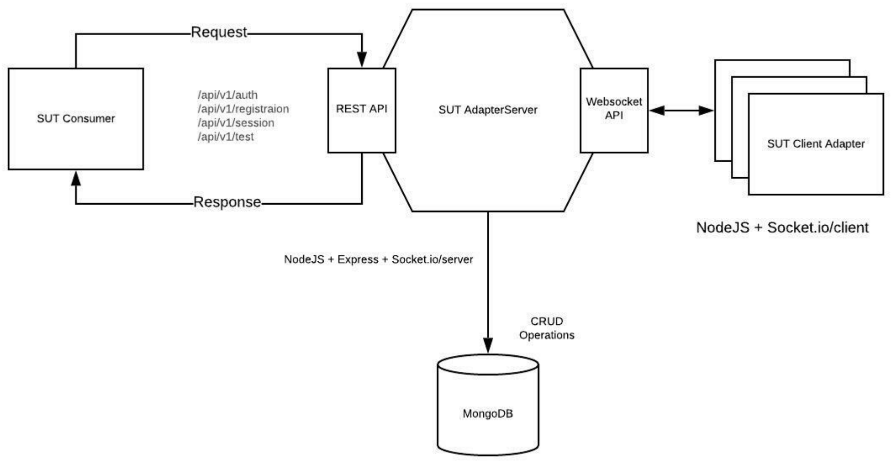

Figure 5.1: High level overview Adapter Framework

The heart of the framework is the SUT AdapterServer that, once running, accepts requests from both SUT Consumers and SUT client Adapters. The SUT Consumer initiates tests against one or more SUT clients using an exposed RESTful API. The SUT Client accepts sequences of events from the SUT Consumer via the server, executes this sequence and sends response to the SUT consumer. The following figure shows the Adapter Framework. In the following subsections, all elements will be discussed in more detail.

Adapter Framework technology stack
This section lists the technology stack that is used to implement the Adapter Framework with a short rationale behind each choice.

- **NodeJS framework usage**
  At least the Server part and the Web Client Adapter are implemented using the NodeJS framework. There are several reasons for choosing a NodeJS implementation. NodeJS is a popular environment to build cross platform applications using JavaScript. Cross platform integrations between SUT consumers and different types of SUT clients should theoretically be achieved easily. Secondly, the SUT Web Adapter Client is based on Puppeteer and Puppeteer natively provides an API that can be accessed using NodeJS.

- **Puppeteer on the SUT Web Client**
  The Web Client Adapter implementation uses Puppeteer to interact with the SUT. The reason for this implementation choice is the fact that Puppeteer natively runs on NodeJS and the NodeJS environment is used for the adapter server. Besides this, Puppeteer can run headless and does not need additional libraries to achieve a headless integration with Chrome web browser which makes interaction with Web applications straight forward.

- **RESTful API**
  The SUT Server exposes a RESTful API to SUT Consumers. The reason behind this is answered by the first research question and discussed in section Generic API.

- **WebSocket SUT Client/Server interaction**
  The SUT Server and SUT Client(s) integrate using WebSockets. Using a WebSockets communication channel between SUT Clients and the server enables real-time, bidirectional and event-based communication. Using the Socket.IO NodeJS library provides additional functionality like Broadcasting, which makes it easy to send events to all the connected SUT Clients, and Rooms which can be used to broadcast events to a subset of SUT clients. These features provide parallel execution of multiple SUT Clients initiated by only one SUT Consumer.

- **Token based Authentication**
  To offer a lightweight Authentication between SUT Consumers, Clients and the Server, the usage of a Bearer token is used. The least needed security can be achieved using this security mechanism. This mechanism is also the most commonly used technique to shield RESTful APIs.

The SUT Consumer

The SUT consumer is the part of the Adapter Framework ecosystem that acts as the consumer and uses the Adapter Framework to integrate with different SUTs. SUT Consumer communicates with the Adapter Framework through JSON over HTTP. The Adapter Framework exposes all required functions through an exposed REST API. See the next subsection for more information about this API.

The RESTfull API

The REpresentational State Transfer (REST) software architecture style is used. This allows platform and programming language independence and facilitating the implementation of new data and processing components once SUT specifics needs to be available to the consumer of the SUT.

To make functionality available to SUT Consumers a RESTful API exposes the API that is discussed in section Generic API. Also part of this API is a supporting interface that is also available as a RESTful API to expose all functionality that is needed before testing can be started. Tasks like client registration, authentication and authorization are offered. Section RESTful API Capabilities gives more details about what exact capabilities are available for the SUT Consumer.

The SUT Adapter Server

The Adapter Server is the heart of the Adapter Framework ecosystem. Once this server is up and running, it listens to incoming HTTP traffic from the SUT Consumers or WebSocket requests from SUT Adapter Clients. Before an actual Test can be executed, both the SUT Consumer and the SUT Client must register themselves to the Adapter Server. After successful registration, both the SUT Consumer and Client authenticate themselves before creating a Testsession. If both the SUT Consumer and the SUT Client successfully created a Testsession, a Test can be initiated from the SUT Consumer. A Test can consist of one or more TestSteps that can be used for facilitating a scriptless testing approach or a TestScript that is capable of facilitating a script-based approach.

SUT Client Adapters

For every specific SUT variant a SUT, Client Adapter can be created. The need for creating a specific Client Adapter depends on the specifics of a certain SUT.

Every adapter should be capable of integrating to and from the generic API exposed via the RESTful and WebSocket APIs.

SUT WebApplication Client Adapter
The SUT WebApplication Client Adapter is initially created to address several research questions. This adapter makes it possible to interact with Web Applications using the Puppeteer Node.js library. The SUT WebApplication Client Adapter exposes the same interface every Client Adapter exposes but translates this interface to the SUT Specifics of a WebApplication.

SUT WindowsDesktopApplication Client Adapters
The SUT WindowsDesktopApplication Adapter uses the Microsoft UI Automation API to interact with Desktop applications.

WebSocket API

A WebSocket API is used to integrate SUT Client Adapters with the SUT Adapter Server. Using WebSockets, real-time, bidirectional and event-based communication is accomplished between the Server and the Adapter client. The concept of rooms and namespaces is offered by the chosen Socket.io library that is used to simultaneously interact with multiple SUT Clients while implementing multi tenancy.

Adapter Framework functionality

The Adapter Framework offers functionality to SUT Consumers and SUT Clients and integrates both using a RESTful API on the consumer (SUT Consumer) side and a WebSocket API on the provider (SUT Client) side. The SUT Client Adapters integrate natively with the API provided by the SUT specific operating system. In the following subsections, all are discussed in more detail.

Adapter Framework Core
The Adapter Framework serves as the integration layer between both the SUT Consumers and the SUT Clients. To the SUT Consumers the Adapter Framework exposes a RESTful API that is offered by the Express Node.js library. For every capability, a different express route is implemented. To SUT Clients the Framework serves a WebSocket connection that will wait for Clients to connect. On every connection attempt from a Client, the Server first will try to authenticate the Client based on the given credentials. Once the Client is authenticated, a Token is returned and can be used with subsequent requests. With the provided credentials, a ClientAdapter should be capable of acting on several events that are emitted sequentially by the WebSocket part of the Adapter Framework. The SUT AdapterServer is developed using Node.js as an asynchronous event-driven JavaScript runtime. The Websocket API is implemented using the Node.js Websocket.io library [36].

SUT Consumer facing RESTful API

The Adapter Framework exposes a RESTful API to SUT Consumers. By exposing this API, the infrastructure part of the generic API is offered. This is done by exposing this API using the generic HTTP requests to the Methods GET, PUT, POST and DELETE.

Besides the usage of the HTTP infrastructure, the RESTful API offers a variety of functionality to support the integration between a SUT Consumer and one or more instances of a SUT Client. The following subsections will elaborate on the capabilities that are offered by this API.

Registration

The Registration resource can be created and accessed using this API. A Registration makes it possible to logically connect a SUT Consumer with a SUT Client using an active Session. Having a valid Registration is a pre-condition to be able to create a Test and executing TestSteps against a SUT.

Session

Session management is used to manage sessions by linking SUT Consumers to SUT Clients. Besides a valid Registration, an active Session is a pre-condition.

A Session can be created using the following API call:

```
POST /api/v1/sessions

Body:
{
  "name": "SUT Consumer",
  "type":"SERVER",
  "description": "The Consumer"
}
```

Authorization
Offering an Authorization layer in front of the Framework offers the capability to register both a SUT Client and a SUT Consumer while shielding the core functionality of the Adapter Framework against unauthorized usage. The Framework provides an OAuth 2.0 authorization scheme using token authentication. The following listing itemizes the underlying API capabilities:

- **register**
  Both the SUT Consumer and the SUT Client must be registered first before authentication and authorization can take place, once both start interacting with the Adapter Framework.

  The following API call can be executed to register a SUT Consumer or a SUT Client:

  ```
  POST /api/v1/auth/register

  {
    "name": "UserName",
    "email": "emailaddress",
    "password": "123456"
  }
  ```

  The following response is returned in case of success:

  ```
  {
    "success": true,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ[
        ...]"
  }
  ```

  Part of the response is a Bearer token with an expiry. To access the core functionality provided by the Framework, this token must be send with every subsequent requests.

- **me**
  The following API call retrieves the currently logged in user:

  ```
  POST /api/v1/auth/me
  ```

  The following response will be sent back after successful login:

  ```
  {
      "success": true,
      "data": {
          "role": "SUT",
          "_id": "6062a2852ab8070a3afdda8d",
          "name": "John Doe",
          "email": "johndoe@gmail.com",
          "createdAt": "2021-03-30T04:01:09.025Z",
          "__v": 0
      }
  }
  ```

- **login**
  To login the following API call can be used:

```
POST /api/v1/auth/login
{
  "email": "johndoe@gmail.com",
  "password": "123456"
}
```

  The following response will be send back successfull login:

```
{
    "success": true,
    "token": "eyJhbGci[..]-EemaThMYfKsmE"
}
```

Test
The Test resource provides the most important part of the RESTful API offered by the Adapter Framework. Using this API a Test can be created for a given registration:

```
POST /api/v1/registrations/{registrationid}/tests
{
  "name": "Test SUT xyz2",
  "serverSession": "605acc49f12658d6c3dcff51"
}
```

Listing 5.1: Create Test

TestStep
Once a Test is created for a given Registration, the following API call can be used by the SUT Consumer to execute a TestStep:

```
POST /api/v1/registrations/{registrationid}/tests/{testid}

Body:
{
    "teststepAction": {
        "teststepActionMetaData": {
            "createScreenshot": "true",
            "ignoredTags": []
        },
        "destination": {
            "sutType": "web",
            "uri": "http://books.toscrape.com/"
        },
        "actionSequence": [
            {
              "action": "click",
              "subactions": [
              {
                "name": "selector",
                "value": "<..>"
              }
```

```
                ]
            },
            {
                "action": "wait",
                "subactions": [
                    {
                        "name": "for",
                        "value": "10000"
                    }
                ]
            },
            {
                "action": "screenshot"
            }
        ]
    }
}
```

Listing 5.2: Execute TestStep

WebSocket API

Once the Adapter Framework is running, it listens to WebSocket connections from SUT Client Adapters. The WebSocket part of the Framework is implemented using the Node.js library, socket.io-client on the Client side and socket.io on the Server side, and is capable of using a bi-directional event based communication channel. The following subsections will elaborate on the capabilities that are offered by this API.

Server side WebSocket API Capabilities

The server side WebSocket API is used for SUT Clients to connect/disconnect to the server, register themselves, creating a testsession with a SUT Consumer, and eventually communicating back the results of a TestStep or TestScript. The concept of Rooms is used to address the challenge servicing multiple SUT Clients against one single SUT Consumer. A room is an arbitrary channel that sockets can join and leave. It can be used to broadcast events to a subset of clients [37]. The following paragraphs describe all offered capabilities.

- **connection**
  The connection event makes it possible for Websocket clients to connect to the server. The following listing shows how the server is initialized using the same server object that is already initialized to serve a HTTP server to the RESTful API consumers.

```
const server = app.listen(PORT, console.log(`Server
    running in ${process.env.NODE_ENV} mode on port ${
    PORT}`.green.bold));
const io = socketIO.init(server);

io.use(function (socket, next) {
    authenticate(socket, next);
}).on('connection', onNewWebsocketConnection);
```

Listing 5.3: Websocket Server initialization

- **disconnect**
  If Websocket Clients gracefully or forcefully disconnect from the Server, this event is raised. Once this event is received, the Server inactivates the Websocket Client.

- **create_session**
  If a SUT Client is successfully connected to the server, the create_session event can be emitted by the client to create a session. Given the registration and the socketId of the connected client, the server first tries to activate an inactive or pending session. The server creates a new session if no matching session exists.

- **create_test**
  SUT Clients can create a test by emitting a create_test event. This event should be emitted using a valid registration and session. Once this event is received, the server will first try to find a pending test. If a pending test is found, it means that a test is initiated by a SUT Consumer but a Client has not connected yet. If no pending test is found, a new one is created.

- **create_teststep**
  After creating a test using the create_test event the next step is actually executing a sequence of actions against the SUT by emitting this event. The payload send with this event consists of the data as described in section Generic API.

- **teststep_executed**
  If a teststep is executed by the SUT Client, the client can emit this event to send the results to the server. The server sends the result to the SUT Consumer.

- **testscript_executed**
  If a testscript is executed by the SUT Client, the client can emit this event to send the results to the server. The server sends the result to the SUT Consumer.

- **screenshot**
  The screenshot event can be emitted by the client to send screenshot data to the server. Once the server receives this event, the screenshot is saved in the document store.

Client Adapter side WebSocket API Capabilities

For a Websocket integration between the server and Client Adapters, the following websocket events must be implemented by a Client Adapter:

- **success_login**
  After a connection attempt, the server automatically authenticates the client and logs in. If this succeeds, the server will emit the success_login event. After this event is received, the Client is able to register itself.
- **session_created**
  The session_created event is emitted by the server after activating a pending or inactive session or after creating a new session.
- **execute_teststep**
  This event makes it possible for a Client to, based on the given payload, execute a teststep against the SUT. The payload consists of the datamodel that is described in section Generic API
- **execute_testscript**
  This event makes it possible for a Client to, based on the given payload, execute a testscript against the SUT. The payload consists of the datamodel that is described in section Generic API

The Document-based storage

The document-based storage is used as an object store to persist relevant information about SUT consumers, SUT client adapters and test sessions once test execution takes place. The Document-based storage is a MondoDB Atlas based storage containing the gray shaded objects of the following logical data model:

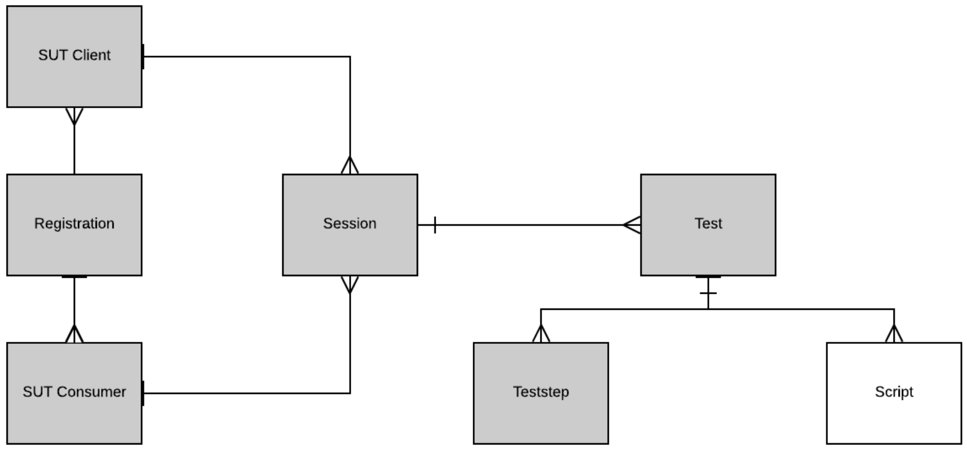

Figure 5.2: Logical Model Document-based storage

- Registration. This schema supports the Registration API capability by linking a SUT Consumer and a SUT Client with one or more Sessions
- Session, By supporting the Session API this schema stores Session information about connected and disconnected SUT Consumers and Clients.
- Test, The Test API is backed using this schema. Using this schema active and inactive tests can be distinguished.
- TestStep, The status of TestSteps are stored in this schema. This schema supports the TestStep
- User, This schema is used to register a SUT Consumer and a Client with valid credentials. The Authorization API is supported using this schema.

Chapter 6

Validation

This section addresses the proposed API combined with the Adapter Framework and answers the following research question:

> **RQ2:** What are the challenges and possible solutions looking at remote and parallel model-based GUI testing?
>
> **and**
>
> **RQ3**: How can the implementation be validated by several consumers?
> Validating the implementation will be done by integrating a script based consumer(e.g. WebDriver) and a scriptless based consumer (e.g. TESTAR).

Both the script-based and scriptless consumers can use the same API and must meet the same pre-conditions (see sections: Authorization, Registration, Session) to be able to initiate a TestStep. Using a content based routing integration pattern on the serverside the server can distinguish which type of integration (scriptless or scriptbased) is needed. For the script-based consumers the distinguishing factor is the root tag of the JSON body "**testscript**" :

```
POST /api/v1/registrations/{registrationId}/tests/{testid}

Body:
{
  "testscript": {
    "code": "<testscriptToExecute>"
  }
}
```

Listing 6.1: script based integration

The scriptless approach can be denoted by using the "**teststepAction**" root tag:

```
POST /api/v1/registrations/{registrationId}/tests/{testid}

Body:

{
    "teststepAction": {
        "teststepActionMetaData": {},
        "destination": {},
        "actionSequence": []
    }
}
```

Listing 6.2: scriptless integration

6.1 Integrating script-based consumers

One important objective this research has shown is the possibility to integrate a script-based consumer via the proposed API with a specific backend SUT. In the controller of the tests.js the following condition applies for a script-based consumer:

```
1  if (req.body.testscript){
2
3    socket_action = 'execute_testscript';
4
5    teststep = {
6      teststepMetadata,
7      testscript: req.body.testscript
8    }
9  }
10
11 socket.to(session.socketId).emit(socket_action, teststep);
```

Listing 6.3: script-based request handling

Based on the JSON Key "testscript" on line 1 in Listing 4.1 the server first determines the socket action (line 3), composes a request (line 5-8) and publishes a Websocket event "execute_testscript" against the socketId of the connected SUT client (line 11).

On the client side, the following implementation applies:

```
socket.on('execute_testscript', function(testscript){

    (async () => {
        console.log('Executing testscript');

        let data = JSON.stringify(testscript.testscript.
            code, null, 2);

        fs.writeFileSync('script.js', data);

        runScript('script.js', function (err) {
```

```
11              if (err) throw err;
12              console.log('finished running script');
13          });
14
15          let testscriptOutcome = {
16              "data":data
17          }
18          console.log('emit testscript_executed');
19
20          socket.emit('testscript_executed',
                testscriptOutcome);
21
22      })().catch((e) => console.error(e))
23  })
```

Listing 6.4: script-based event handling

Line 1 shows the subscribing on the ”execute_testscript” event. After parsing the script file (line 6 and 7), the script is written to a script file (line 8) and the runScript function (line 10) is invoked. The next listing shows the runScript function implementation:

```
1
2   const childProcess = require('child_process');
3
4   function runScript(scriptPath, callback) {
5
6       console.log("In Run Script");
7       // keep track of whether callback has been invoked to
            prevent multiple invocations
8       var invoked = false;
9
10      var process = childProcess.fork(scriptPath);
11
12      // listen for errors as they may prevent the exit event
             from firing
13      process.on('error', function (err) {
14          if (invoked) return;
15          invoked = true;
16          callback(err);
17      });
18
19      // execute the callback once the process has finished
            running
20      process.on('exit', function (code) {
21          if (invoked) return;
22          invoked = true;
23          var err = code === 0 ? null : new Error('exit code
                ' + code);
24          callback(err);
25      });
26  }
```

Listing 6.5: child process script execution

Above listing uses the Node.js child_process module to fork a new Node.js process (line 10) [38]. In the on 'exit' branch (line 20), the results of the script are captured and returned and listing 6.4 composes the event (line 15) and publishes the ”**testscript_executed**” event to the server.

6.2 Integrating scriptless consumers

To integrate a scriptless consumer, the data model showed in listing 6.2 offers the capability to send **teststepActionMetaData** a **destination** and an **actionSequence**. The **tesstepActionMetaData** can be used to give additional instructions to the SUT Client Adapter. The following listing shows a possible teststepActionMetaData:

```
"teststepActionMetaData": {
   "createScreenshot": "true",
   "emitScreenshot": "true",
   "sendScreenshot": "true",
   "ignoredTags": ["script", "noscript", "head", "meta", "
       style", "link", "svg", "canvas"]
}
```

Listing 6.6: teststepActionMetaData

Besides the teststepActionMetaData, also a destination object must be present. The following listing shows the presence of the destination object where the sutType web indicates the usage of a SUT WebClientAdapter backend that will navigate to the given uri:

```
"destination": {
  "sutType": "web",
  "uri": "http://books.toscrape.com/"
}
```

Listing 6.7: destination

The given actionSequence makes it possible to perform actions against the given SUT. The following listing shows a possible action sequence that will be sent to the SUT WebClientAdapter to be interpreted and translated to native SUT specific commands.

```
"actionSequence": [
    {
       "action": "click",
       "subactions": [
          {
            "name": "selector",
            "value": "#default > div > div > div > div >
                section > div:nth-child(2) > ol > li:nth-
                child(1) > article > div.image_container >
                a > img"
          }
        ]
    },
    {
       "action": "wait",
       "subactions": [
           {
                "name": "for",
                "value": "10000"
            }
        ]
    },
    {
     "action": "screenshot"
    }
]
```

Listing 6.8: destination

A WebAppClientAdapter written in NodeJS listens to the **execute_teststep** websocket event. The adapter uses the Puppeteer API to translate the given actionSequence to instructions against the destination uri. The following code listing shows a possible implementation of how the translation from the generic actions are translated to more specific Puppeteer API calls:

```
for (var actionSequence in teststep.teststepAction.
    actionSequence)
{
  let actionType = teststep.teststepAction.actionSequence[
      actionSequence].action;

  if (actionType == 'click') {
    let subaction = teststep.teststepAction.actionSequence[
        actionSequence].subactions[0]
        await page.click(subaction.value)
  }

  if (actionType == 'keyboard') {
    for (subaction in teststep.teststepAction.
        actionSequence[actionSequence].subactions)
    {

      subactionName = teststep.teststepAction.
          actionSequence[actionSequence].subactions[
          subaction].name;

      subactionValue = teststep.teststepAction.
          actionSequence[actionSequence].subactions[
          subaction].value;

      if (subactionName == 'type') {
        await page.keyboard.type(subactionValue);
      }
      if (subactionName == 'press') {
        await page.keyboard.press(subactionValue);
      }
    }
  }
 }
```

Listing 6.9: API mapping to Puppeteer

Java based SUT Consumer

Part of the research was to integrate a scriptless SUT Consumer with a SUT Client Adapter using the proposed API. One of those consumers is TESTAR. Because TESTAR is written in Java and will act as a Java based consumer, a prototype using the Jackson Java JSON library has been created. Using this prototype the RESTful API can successfully be consumed.
Using Plain Old Java Objects (POJO) the Jaxon API can be used to interact with the exposed RESTful API. For instance, using the UserRegistration domain object, a registration can be implemented as following:

```java
public UserRegistration register(UserRegistration
    userRegistration) throws Exception {


  WebTarget registerWebTarget = webTarget.path("/api/v1/
      auth/register");

  Invocation.Builder invocationBuilder =
      registerWebTarget.request(MediaType.
      APPLICATION_JSON);

  Response response = invocationBuilder.post(Entity.
      entity(userRegistration, MediaType.APPLICATION_JSON
      ));

  UserRegistration registrationResult = response.
      readEntity(UserRegistration.class);

  if (!registrationResult.isSuccess()) {
    throw new Exception(registrationResult.getError());
  }

  return registrationResult;
}
```

Listing 6.10: User Registration against RESTfull API using Jaxon

Once all the pre-conditions (see sections: Authorization, Registration, Session) are met, a scriptless TestStep can be executed as following:

```java

public void executeTestStep(String token, String
    registrationId, String testId, TestStep testStep) {

  javax.ws.rs.core.Feature feature = OAuth2ClientSupport.
      feature(token);
  client = ClientBuilder.newBuilder().register(feature).
      build();

  Response response = client.target("http://localhost
      :5000/api/v1").path("/registrations/"+
      registrationId+"/tests/"+testId).request()
      .property(OAuth2ClientSupport.
          OAUTH2_PROPERTY_ACCESS_TOKEN, token).post(
          Entity.entity(testStep, MediaType.
          APPLICATION_JSON));

  }
```
Listing 6.11: Execute scriptless TestStep against RESTfull API using Jaxon

Chapter 7

Results and conclusions

With emphasis on what information is needed to extract the state tree, locating elements on the GUI and executing a sequence of actions against the GUI of a Web based and Windows Desktop applications, a remarkable amount of overlap can be seen between the analysed APIs. Analysis 4.2 shows that **provided APIs are very similar and overlap a lot**. This analysis has been executed to give an answer to the first research question (RQ1) and its sub questions.
To uniquely identify elements on the GUI, the same element attributes (id, name or xpath) can be used. Advanced locator strategies, such as CSS selector or xpath, are both capable to find almost any HTML element on a web page. Additionally, for a Web based SUT that is styled with a CSS, a CSS Selector can be used. Besides these properties, also ARIA Widget properties (2.1) can be part of the DOM tree of an Web application. Many ARIA widgets are currently incorporated into HTML5, making this standard not usable anymore. To locate elements on a Desktop Application, using the Windows Automation API, the AutomationId of an element can be used and combined with indications about what kind of control pattern is available for a given AutomationElement. While populating the state tree of a SUT, this information can be abstracted away where the filtered set of O only contains the actionable widgets of the SUT and a unique identification of the widget combined with the type of widget.
An adapter framework has been written to implement the proposed API while giving answers to the second research question (RQ2) and its sub questions. The selected technology stack (Adapter Framework technology stack) offers many functionality and helped answering these sub questions. The NodeJS ecosystem offers a high number of libraries that can easily be used to, for example expose a RESTful API to SUT Consumers, or integrate with a MongoDB document store. The Express framework is a minimal and flexible NodeJS web application framework that provides a robust set of features for web and mobile applications. The mongoose NodeJS library gives mongodb object modeling capabilities for NodeJS and supports the writen framework by offering a persistence layer to store information about executed teststeps and screenshots that are captured during testexecution.

To answer the first research question (RQ1) and its sub questions (RQ1.1 - RQ 1.3), an overlap analysis has been executed to find the differences and commonalities between web based programmable APIs and the Windows Desktop Accessibility API. The goal of this analysis is proposing an API that can be used to adapt to multiple SUT types (e.g. Desktop, Web, native Apps). The results obtained from this analysis gives information about the degree of overlap between these APIs and is based on the identified 3 categories (extracting state, locating elements and executing actions). This analysis uses a black box approach analysing the available APIs. The absence of a more in depth analysis of the behaviour of programable APIs and the SUT and the absence of analysing native Apps is a threat to the validity of the proposed API. Another threat to the validity of the crafted framework is the absence of applying a "real" integration between e.g. TESTAR in the role of a SUT consumer and the used SUT WebClientAdapter as the providing Adapter part that interacts with a web based application. Furthermore to be able to validate the proposed solution as a whole a SUT DesktopClientAdapter should implement the proposed API. In the background section the Webdriver specification has been discussed where Appium, Selenium Webdriver and WebdriverIO are outlined as implementing frameworks. However these frameworks have not been used to implement the Generic API and parts of the Adapter Framework.

If a more generic API acts as an intermediate layer, a translation must be made from the client specifics to a more generic API and vice versa. This has been demonstrated with a draft implementation of a Java based SUT Consumer. This translation comes with an initial integration effort for each SUT type and a possible maintenance cycle once the SUT specific API starts to evolve to a newer specification, where parts of the old specification gets deprecated for instance. It is not measured how much the integration effort is or how much effort is needed to upgrade the SUT Adapters to a newer version once this version is available.

Chapter 8

Personal Reflection

Pursuing this master degree has been one of my highest goals for the past years. I knew upfront that this study and especially this part, the graduation assignment, will be hard. The topic of this thesis attracted me and I was fully motivated and started off. I dived into new technologies and learned a lot about new concepts, programming languages and frameworks. During the process many things changed in my personal live including becoming a father multiple times, unfolding my professional career and moving to a newly build home. Because of these life changing events I was not able to give all my energy and time finalizing this study the way I wanted to finalize it in the first place. Nevertheless while looking back, I proudly can say, yes did it!

I want to give a special thank to Pekka for his patience as my supervisor and sparring partner. Thank you for your input and trust and pulling me through the whole process! I also want to thank Tanja as my supervisor for catching my interest in the topic of GUI testing using TESTAR. My last words go to my wife Monique, she always gave me the space and trust I needed during this study as a whole and in particular this graduation assignment. I can not thank you enough for your patience and mental support. Thanks you!!!

Bibliography

[1] Antonia Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[2] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, March 2014.

[3] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and Evolving GUI-directed Test Scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Emil Alégroth and Robert Feldt. On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering*, 22(6):2937–2971, December 2017.

[5] Tanja E. J. Vos, Peter M. Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. TESTAR: Tool Support for Test Automation at the User Interface Level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.

[6] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. WebMate: Generating Test Cases for Web 2.0. In Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann, editors, *Software Quality. Increasing Value in Software and Systems Development*, Lecture Notes in Business Information Processing, pages 55–69. Springer Berlin Heidelberg, 2013.

[7] Atif M. Memon and Mary Lou Soffa. Regression Testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 118–127, New York, NY, USA, 2003. ACM.

[8] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679–1694, October 2013.

[9] Anna I. Esparcia-Alcázar, Francisco Almenar, Tanja E. J. Vos, and Urko Rueda. Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool. *Memetic Computing*, pages 1–9, June 2018.

[10] Gigon Bae, Gregg Rothermel, and Doo-Hwan Bae. Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study. *Journal of Systems and Software*, 97:15–46, November 2014.

[11] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. AutoBlackTest: A Tool for Automatic Black-box Testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1013–1015, New York, NY, USA, 2011. ACM. event-place: Waikiki, Honolulu, HI, USA.

[12] Stanislava Nedyalkova and Jorge Bernardino. Open Source Capture and Replay Tools Comparison. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 117–119, New York, NY, USA, 2013. ACM. event-place: Porto, Portugal.

[13] Weiran Yang, Zhenyu Chen, Zebao Gao, Yunxiao Zou, and Xiaoran Xu. GUI testing assisted by human knowledge: Random vs. functional. *Journal of Systems and Software*, 89:76–86, March 2014.

[14] Amira Ali, Huda Amin Maghawry, and Nagwa Badr. Automated parallel GUI testing as a service for mobile applications. *Journal of Software: Evolution and Process*, 30(10):e1963, 2018.

[15] Selenium.

[16] Appium.

[17] W3C. WebDriver W3C Working Draft, August 2020.

[18] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Automatic testing of GUI-based applications. *Software Testing, Verification and Reliability*, 24(5):341–366, 2014.

[19] Urko Rueda, Tanja E J Vos, Francisco Almenar, Mirella Oreto, and Esparcia Alcazar. TESTAR - from academic protoype towards an industry-ready tool for automated testing at the User Interface level. page 5, October 2016.

[20] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Robula+: an algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, March 2016.

[21] Atif Memon, Adithya Nagarajan, and Qing Xie. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(1):27–64, 2005.

[22] João Carlos Silva, João Saraiva, and José Creissac Campos. A Generic Library for GUI Reasoning and Testing. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 121–128, New York, NY, USA, 2009. ACM.

[23] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Pesto: Automated migration of DOM-based Web tests towards the visual approach. *Software Testing, Verification and Reliability*, 28(4):e1665, June 2018.

[24] Francisco Almenar, Anna I. Esparcia-Alcázar, Mirella Martínez, and Urko Rueda. Automated Testing of Web Applications with TESTAR. In *Search Based Software Engineering*, Lecture Notes in Computer Science, pages 218–223. Springer, Cham, October 2016.

[25] Andres Gonzalez and Loretta Guarino Reid. Platform-independent Accessibility API: Accessible Document Object Model. In *Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*, W4A '05, pages 63–71, New York, NY, USA, 2005. ACM.

[26] Microsoft. UI Automation Specification, 2018.

[27] W3C. Accessible Rich Internet Applications (WAI-ARIA) 1.1, December 2017.

[28] W3C. HTML Accessibility API Mappings 1.0, August 2020.

[29] W3C. W3C DOM 4.1, March 2020.

[30] Mehdi Mujtaba. selenium javascript automation testing tutorial for beginners, June 2020.

[31] Hans Manoj. *Appium Essentials*. Packt Publishing, 2015.

[32] Puppeteer, 2021.

[33] W3C. WADL, August 2009.

[34] json-schema org. JSON Schema, February 2021.

[35] Roy Fielding. Representational State Transfer (REST).

[36] socket.io. socket.io, February 2021.

[37] socket.io. socket.io/rooms, February 2021.

[38] Nodejs. NodeJS_child_process, July 2021.

List of Figures