Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Report-Video Production for Quick Bug-finding in the Web Applications

Erstellung des Report-Videos für Schnelle Fehlerfindung in den Web-Applikationen

Bachelor's Thesis

im Studiengang Computer Science

von

Malvin Andika Tandun

Prüfer: Prof. Dr. rer. nat. Kurt Schneider
Zweitprüfer: Dr. Jil Ann-Christin Klünder
Betreuer: M. Sc. Jianwei Shi

Hannover, 20.01.2022

Declaration of Authorship

I hereby declare that the Bachelor's Thesis is my unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources, I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here. This paper was not previously presented to another examination board and has not been published.

Hannover, 20.01.2022

Malvin Andika Tandun

Abstract

Report-Video Production for Quick Bug-finding in the Web Applications

To test web applications' performance, regression tests are done. These tests can be done by interacting with the Graphical User Interface (GUI) of the web application. Regression tests are done periodically and provide the user with information about a web application's performance over time. This information is helpful to check if a new update affects the application's performance.

These tests are time consuming and doing these tests manually would cost a lot of resources. To solve this problem, automated regression tests are made. Selenium WebDriver is a very popular test automation software that provides users with the ability to interact and manipulate a web application's GUI. These automated tests could be run overnight and results could be checked the following day.

If there are failed tests, users could investigate the causes of these failures to debug problems. Selenium WebDriver doesn't provide a way to track the source of these failures and third party applications are needed to do the logging work of these tests The focus of this project is to provide Selenium users with a way to document their tests by providing a report in video form by using ScreenTracer. The goal of this report is to allow the user to use ScreenTracer to find problems quickly.

Zusammenfassung

Erstellung des Report-Videos für Schnelle Fehlerfindung in den Web-Applikationen

Um die Leistung von Web-Applikationen zu testen, werden Regressionsteste ausgeführt. Dieser test könnte in Form eines GUI Tests ausgeführt werden, indem man durch die Web-Applikationen mithilfe eines Mauses und eines Tastaturs navigiert. In der Softwareindustrie soll jede Funktionalität getestet werden, bevor die zum Publikum vöffentlicht wird. Dieser Regressiontest dient dazu, Bugs zu finden und Performanz zu testen, damit die Entwickler sicher sein können, dass die Funktionalität gut ist, bevor sie veröffentlicht wird.

Diese Teste könnte manual ausgeführt werden, aber es ist sehr zeitaufwändig. Vor allem in der heutigen Industrie, wo Funktionalitäten regulär hinzugefügt werden. Automatisierte Test erleichtet dieses Problem. Selenium WebDriver ist ein Gerät, womit man die Teste, vor allem GUI Teste, automatisieren kann. Diese Teste könnte automatisch nach einer bestimmten Zeitplan ausgeführt werden z.b. jede Nacht.

Falls die Teste durchgefallen sind, könnte die Entwickler nach der Fehler suchen. Das hier ermöglich regüläre umfassende Teste. Das Problem liegt daran, dass nach jedem Test würde so ein "Protokoll" erzeugt, dass Informationen über den Test beinhaltet. Dieser Protokoll könnte Screenshots beinthaltet, Textbeschreibungen, usw. Es könnte sein, dass diese Informationen nicht genug sein, dass verursacht, dass Bugs nicht gefunden werden, oder dass die Untersuchung des Bugs viel zu viel Zeit aufnimmt.

In dieser Arbeit, wird die Software ScreenTracer eine Lösung anbieten, Videos als Bug Reports zu benutzen. Es wird mehrere Funktionalitäten eingebaut, dass diese Software verbessert. Das Ziel ist ScreenTracer zu verbessern damit Software Entwickler schneller Bugs finden können. Nachdem die Anforderungen erfüllt sind, wurde eine Benutzerstudie durchgeführt, um die Geschwindigkeitsverbesserung beim Debuggen zu untersuchen.

Contents

Chapter 1

Introduction

Due to the rapid development of web apps, developers update their applications more frequently. Regression tests check if an update affects the application's performance. This shorter development cycle means more testing. Automated tests save a lot of human power needed to do manual tests, especially if these tests are repeatable.

According to a survey conducted in 2018 by Tricentis and Techwell [10], only 7% of participants did not automate any of their tests. This number was to expected to drop to 3% in the following year. Among the 173 participants, 73% of them reported using Selenium as their automation framework. Less than half (41%) of them said that test automation could help catch bugs earlier.

From these results, we can conclude that test automation is growing and Selenium is one of the most popular frameworks used. Despite the numerous advantages of test automation, its capability to find bugs is not optimal. This problem can be traced to the lack of logging capabilities of Selenium WebDriver. Although automated tests could detect errors, tracing where they come from by investigating the source code is not an easy task.

ScreenTracer allows users to record their automated tests. In cases of failures, ScreenTracer enables users to find problem areas quickly by viewing the replay on the ScreenTracerViewer. ScreenTracerViewer provides users with the needed information to debug errors by navigating its Graphical User Interface (GUI).

1.1 Research Aims and Objectives

The current state of ScreenTracer enables users to record and view their recordings using ScreenTracerViewer. Developers that test their web applications can find problem areas by navigating the Graphical User Interface of ScreenTracerViewer. ScreenTracerViewer lacks fundamental requirements that would satisfy the customers and the design of the

implemented functions are not intuitive.

The main goal of this project is to improve ScreenTracer and ScreenTracerViewer. There are two parts on how to improve this software.

The first part is to implement requirements that would bring user satisfaction substantially higher. Requirement Engineering is needed to determine which requirements are more important than others, and which are realistic to implement.

It is desirable to have an intuitive software interface that enables users to understand the software quickly and navigate through the Graphical User Interface quickly. The second part is to use a scientific approach to GUI design to objectively measure if there are any noticeable improvements.

1.2 Structure of the thesis

This thesis is divided into 6 chapters. The first chapter introduces readers to the goals and objectives of this paper and gives readers a brief summary of how this work is structured. The second chapter lists the related works that this paper is based on and explains the basic fundamentals. The third chapter lists all main and optional requirements that are set for ScreenTracer during the development phase. The fourth chapter gives an overview of how all the main requirements are made, including back end and front end implementation, and explains why the author prefer certain solution to implement a requirement rather than others. Chapter 5 describes the user study that is done in detail, including how the user study is made, how it is executed and the analysis of its result. The last chapter reviews the work and adds ideas on how future work can be done to improve ScreenTracer further.

Chapter 2

Fundamentals

In this chapter works that are related to this paper are listed and discussed. These works would be the base of the fundamental chapter, which explains key themes to this work. These themes are automated testing, ScreenTracer, bug reports and SeleniumWebdriver.

2.1 Related Works

Test automation is a well-researched theme and attempts to improve this topic has been done over the years. In this section, the numerous works and researches that contribute to this thesis are described.

2.1.1 Video based process tracking for systems with GUIs

Holzmann [1] created ScreenTracer to track GUI automated tests by using videos. This work explains the fundamentals of creating the software. It discusses the ideas, the implementations and detailed explanations on ScreenTracer.

2.1.2 What makes a good bug report

Zimmerman [12] did a user study on bug reports and showed factors that are important to make a good bug report. It lists the importance of every type of content and takes notes on which content are the hardest to implement.

2.1.3 GUI Testing : Pitfalls and processes

Memon et. al. [5] describes the process of GUI tests. The tools that are used to test GUIs and the weaknesses of these tools are clarified.

2.1.4 The evolution of test automation

A survey done by Techwell and Tricentis [10] describes the continuous and rapid development of test automation. In this survey statistics on the development of test automation in companies are shown. These include the type of frameworks, the ratio of automated test cases, and future plans for test automation in companies.

2.2 Automated Testing

In a development cycle of software, to ensure quality, procedures such as requirement engineering, design analysis, testing are done. This can also be seen in the widely-used waterfall model that is credited to Royce [9]. And although the agile methodology is widely used, the steps that are done in this model are not completely changed, and this includes testing.

According to Hunt [2], agile methodology acknowledges that user requirements change and it prioritizes the software development. It responds to change and allows frequent and regular software releases. Softwares have to be tested before release and regular releases mean more tests have to be done. Doing tests regularly ensure developer that every feature added is working properly and bugs that are found can be responded to quickly.

Testing the software manually requires a lot of human power and in some cases would be impossible. E.g. trying to overload the program by clicking a button multiple times a second. In this test, a human would be inaccurate or would be too slow. Or checking a list of items in a shopping application that every item's price is displayed correctly; would be manually possible but this would take too much time.

Automated tests are done to save the human resources required to do such tests. And after each update on the software, these tests could be rerun. Developers can do these tests nightly to catch on bugs quickly, and bug reports are made in case of failed test cases. These bug reports could include screenshots, steps taken to create the bug, exceptions thrown, etc.

2.3 ScreenTracer

ScreenTracer is a project by H. Holzmann [1]. This software consists of two components, the ScreenTracer and ScreenTracerViewer. ScreenTracer records a selenium test case and saves it as a file in ".stc" format. The ScreenTracerViewer plays ".stc" files as video. ScreenTracer records a selenium test case and saves it as a file in ".stc" format. The ScreenTracerViewer plays ".stc" files as video.

The ScreenTracerViewer has some functions that support the user to navigate through the video. There is the video bar, which displays the

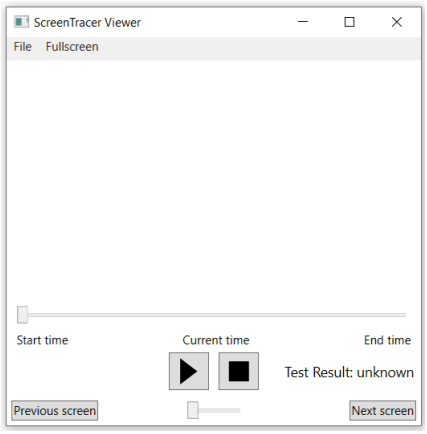

Figure 2.1: ScreenTracerViewer

current time of the video. The user can click on this bar to navigate to previous or next scenes. There is a text that displays the current time of the video. The play and the stop button can be used to pause and resume a video. At the end of the video, the stop button can be used to rewind the video. On their right side, the result of the tests will be displayed, either a passed or a failed test. On the bottom of the screen, there are buttons to navigate to the previous screen and next screen, which changes the current scene to the next or previous 'keyframe'. There is also a slider to speed up the playing video. This slider has a value from 1 to 10, which each number represents the playing speed of the video. E.g. If the slider is on the 2nd level, meaning having a value of '2', the video will be played twice as fast as normal.

Unlike traditional video files, ".stc" are made of frames and keyframes. ScreenTracer doesn't capture the whole screen every specified time tick. It instead waits for a change in the screen, then captures the area on which the change has occurred. This newly captured area will be displayed on top of the previous one as a single 'frame'. If a change is bigger than the previous change, it will then be detected as a 'key frame'. When a 'key frame' is detected, ScreenTracer will capture the whole screen. 'Keyframes' are also taken after a maximum number of frames taken in reached, to prevent too much loss of information. By using this concept, ".stc" files are smaller than traditional video files. A more detailed explanation is provided in a paper by Holzmann et. al. [8].

2.4 Bug Report

A study by Zimmerman et. al. [12] discusses factors that make a bug report good. One of the factors is the content of the bug report. The top three most

important contents listed by developers are *steps to reproduce*,with 78% of developers voting it as important, *test cases* with 43% and *stack traces* with 33%.

Though its importance, *steps to reproduce* is one of the hardest items to document. It is also stated, that incomplete information in *steps to reproduce* is the most severe problem in bug reports. In this study, in question D5, developers are encouraged to share their free thoughts, and interestingly, many comments saying *"The biggest causes of delay are not wrong information, but absent information."*

According to Atif M. Memon [5], one of the most difficult things in finding bugs in GUI are problems that happen between test points.This happens because test automation does not verify all elements after each interaction, errors that happen between steps could be hard to detect.

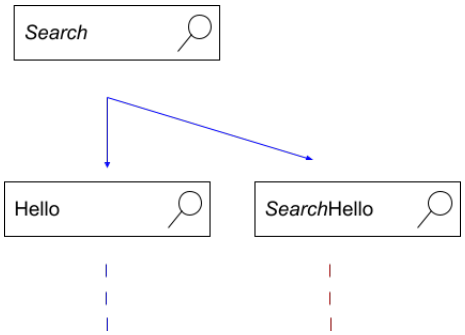

Figure 2.2: Intermediate Error

For example, if there is a mistake on a non-critical GUI Element that does not change the end result. In Figure 2.2 a text box that takes the user input is shown. If the text box function was implemented correctly, the text "Search" inside the text box will be overwritten by the user input. In this example, it isn't implemented correctly and the text box will continue searching with the wrong user input. Depending on how the test automation is implemented, this error could be undetected until the end of the test. In the worst-case scenario, this error would go undetected if there were no tests to check if the input string was correct.

These two studies share a conclusion, that the completeness of a bug report is important. A video format would provide this. An argument against video format is its size. But as explained by Holzmann et. al. [8], ".stc" files are smaller than traditional video files, and even if it takes more space than screenshots, the gain in information justifies its size increase.

2.5 Selenium WebDriver

Selenium WebDriver is an API(Application Programming Interface) that drives a browser natively or remotely. It enables developers to automate tests that are browser-based. It provides functions to navigate the browser and to interact with the Web Elements that are displayed on the browser.

Selenium WebDriver users can interact with the web pages by navigating to the web elements by "finding" them and then sending the commands that they wish to do. E.g. in the google homepage, there is a search box where users can type their text. To do this, first users have to find this element one of the selectors given by Selenium WebDriver (XPath, CSS Selector, ID to name a few). On the right side of the figure below, the "inspect element" function of google chrome is shown. Using this users can find the XPath, ID or CSS locator of web elements.

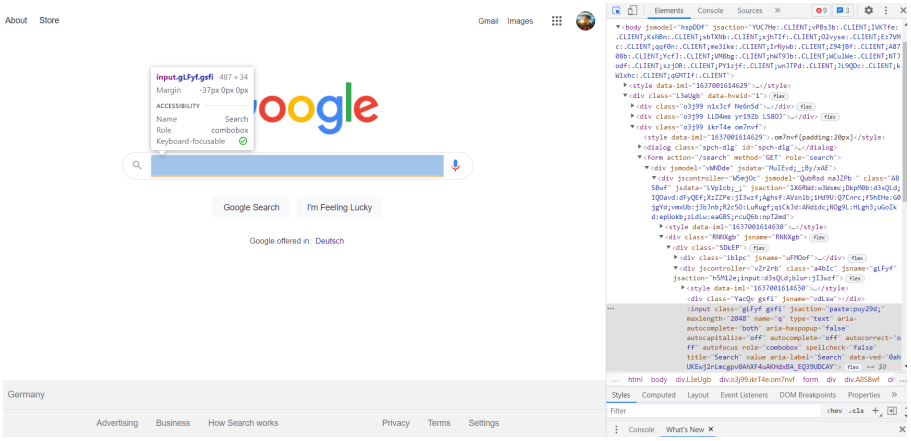

Figure 2.3: Google homepage and its DOM Tree

If the element is successfully found, users can either use the provided functions, which include clicking and sending keys or invoke javascript on this element. There are also functions to check if web elements are clickable, are enabled or are visible to the users.

Chapter 3

Requirements

This chapter explains both the mandatory and optional requirements. The problems which lead to each requirement are to be clarified. In Chapter 3.1 the design process is further explained. Chapter 3.2 states the stakeholders. Chapter 3.3 lists the mandatory requirements and Chapter 3.3 lists the optional requirements.

3.1 Design process

The design process is based on the agile methodology. After gathering the requirements, a weekly plan is created by using Google Sheets. The plan provides a deadline for each subtask and each subtask is divided into smaller tasks in form of cards. These cards are then put on a board resembling the Kanban board.

Each card has a difficulty level from one to four with one representing the easiest and four the hardest. It also represents the estimated number of Pomodoro sessions needed to finish the single task. Tasks that take more than four Pomodoro sessions should be broken down into smaller tasks. Each Pomodoro session consists of 50 minutes of working and 10 minutes pause. In a single day, a maximum of four Pomodoro sessions can be done. Taking this into account, a maximum of twenty eight points can be done in a single week.

Every week a meeting is held with the supervisor to discuss the progress made for the work, the problems that are encountered, and ideas to improve the current ScreenTracer software. These are the basis to create new tasks to be put on the Kanban board. This system ensures productivity and enables the measuring of a workweek. To guarantee consistent code, the Microsoft C# Coding convention [6] will be used.

3.2 Stakeholders

The stakeholders are test automation engineers that need to implement a logging system in their test automation, or that want to record their automated tests. The recorded test cases are suitable for reports because users can embed them into them and they are lighter than the traditional video file. The automated test cases also allow stakeholders to find bugs quickly in their software.

3.3 Mandatory Requirements

R1 *When a video file is open in ScreenTracerViewer, the user can change the video playback speed.*
The test automation done by Selenium WebDriver can be too fast for the viewer to follow.

R2 *Before starting a recording session, the user can select the recording area.*
ScreenTracer records the whole screen, which is inefficient. It should only capture the area where the browser is operating in.

R3 *When a video file is open in ScreenTracerViewer, the user can view the standard output from the recorded test.*
Videos that are played by ScreenTracerViewer have important comments in them. Using this function can help the user to log important information.

R4 *During recording, ScreenTracer highlights GUI Elements that the Selenium WebDriver are interacting with.*
Selenium WebDriver highlight GUI Elements automatically.

3.4 Optional Requirements

OR1 *Users are able to cancel a recording session.*

OR2 *ScreenTracer does not record before the Chrome Browser opens and after it closes.*

OR3 *ScreenTracerViewer GUI Redesign to implement new functionalities.*

OR4 *The selected area is saved as preset.*

OR5 *The next screen and previous screen function of ScreenTracerViewer must load under 1 second.*

OR6 *ScreenTracerViewer has a control panel in full screen mode.*

OR7 *ScreenTracer can select the project to record from the GUI.*

OR8 *ScreenTracerViewer is implemented using the MVVM Model for future development.*

OR9 *The previous and next screen button can be held down to skip multiple screens.*

Chapter 4

Implementation

This chapter will clarify the main idea and both the front-end and back-end approaches that were chosen for each main requirement. For some requirements, that are different approaches that can be used to satisfy them. Why certain approaches are more preferable to others are explained as well as their positive and negative aspects. This chapter is intended not only to explain how the code in ScreenTracer works but to provide an illustration of how did the writer came to certain ideas and why some ideas could not be implemented. The previous implementation will be compared to the current implementation. The current implementation represents newly added functionalities into ScreenTracer.

4.1 Design considerations

Before adding any functionality, design considerations are done to ensure that the feature made will be understandable by the user. This is mostly a front-end aspect in development because most back-end aspects are handled without any need for user interaction. In this process certain points are addressed:

- The user can guess how certain aspects of the software works without having used the software
- The user has enough freedom that it does not hinder the workflow
- The user has constraints to prevent the software from breaking

The look and feel of the new functionalities will be based on these main points. The purpose of this design consideration is to allow ScreenTracer users to learn to use the software fast, while also adding a more modern look to the software. To implement the first point correctly, further research into intuitive software engineering is required. Because this is not the main point of this paper, designs that are used are similar to those used in

popular software. For example, for R1, changing the speed of a video is not new functionality. It has been implemented by multiple video players and designing this requirement based on some of the most-used video player applications would introduce familiarity and therefore fulfil the first point.

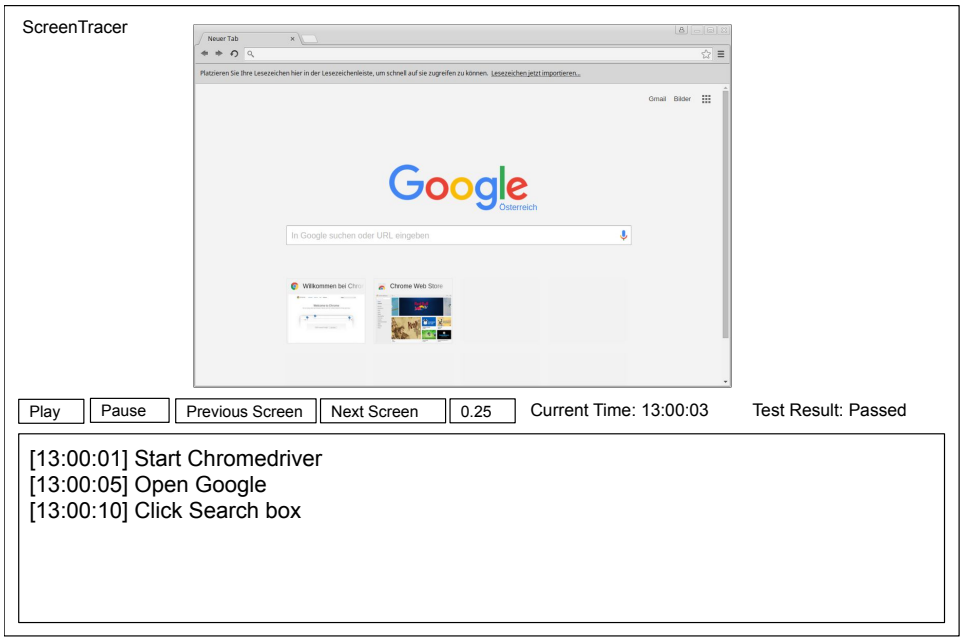

Figure 4.1: Updated ScreenTracer

Figure 4.1 shows the new design mock-up after all the requirements are implemented. The design chosen for this is based on the design considerations pointed above.

4.2 R1: Changing playback speed

Motivation

In Selenium WebDriver actions like clicks and sending text into a web element are done after the said element is found. This process of finding and interacting can be done in a fraction of seconds. This is a huge benefit of using Selenium to automate tests in comparison to manual testing. But this can be problematic when said tests are recorded. Selenium often does this so fast that it is very hard to identify what is happening in the recording. A solution to this would be providing a way for ScreenTracer users to slow down the video playback speed.

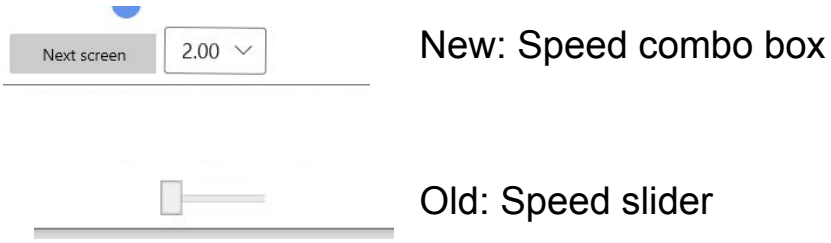

Figure 4.2: Old and new speed function

Previous implementation

ScreenTracer has a slide to adjust the speed of the video which can be seen in figure 4.2. This slider can have a value between one to ten. Changing this value to one means the video is playing at real-time speed, two means twice as fast as the real-time speed, and so forth. The slider is not annotated and it is not clear how much the video speed is annotated just by looking at it. To understand it a user has to try it out a few times by themselves. This also did not allow slowing down the video playback.

Current implementation

Before implementing the code, the design consideration has to be made to choose the most suitable design to present the functionality behind it. After design consideration is done, a combo box is chosen to be the most suitable for this function. A combo box lets the user select a value from limited given values. Users can change the speed anytime without having to pause the video. This application of combo box is also similar to software like YouTube and introduces familiarity to this design. This combo box can be seen in figure 4.2. This combo box support fractional values, these are limited to 0.25, 0.50, 1.00 and 2.00. These number represents how fast the video has to be played in comparison to its real-time counterpart.

To understand the implementation of this function into the code, it is needed to understand, how ScreenTracer plays a video. ScreenTracer takes pictures only when actions happen, so the player needs to simulate the "waiting" that happens in real-time. Every screenshot that was taken also has the time information alongside it. ScreenTracer displays an image, simulate the time waited, then if change happens, it either displays the change on top of the current image or replaces it with a new image (in case of a keyframe).

ScreenTracer creates a thread that waits as long as needed in order to simulate this wait time. For example, if an image should be shown for 1

second, ScreenTracer creates a new thread that waits for 1 second, then terminate itself. ScreenTracer changes the image, and create a new thread again. The duration of wait is calculated constantly by checking the system time, this allows changing the waiting duration while the user is playing the video.

The duration is calculated by measuring the difference of time between two frames and then using the current system time to know when the next frame should be displayed. For example, we have a frame taken at 14:00:01 and the next frame would come at 14:00:03, which means the frame should be displayed for two seconds. In a while loop, ScreenTracer checks if the system time has waited two seconds, by calculating the time that has passed.

Using the acceleration value, ScreenTracer modifies the amount of time that has passed. If the acceleration value is 2.0, the time passed would be multiplied. If the system has waited for one second, then in ScreenTracer the time passed would be counted as two seconds. Enabling to slow down the playback isn't that complicated once the play function is understood. Instead of multiplying the time passed, dividing it simulates the effect of slowing down the video. For example, by changing the acceleration value to 0.5, ScreenTracer divides the time passed in real life. If the system has waited for two seconds, ScreenTracer would multiply this two seconds by 0.5, and act as if only one second has passed.

4.3 R2: Selecting recording area

Motivation

ScreenTracer records the whole screen and captures unneeded information that wastes storage space. This is a big problem because ScreenTracer takes a picture for every change in action, and this includes actions that happen outside the area of interest. During a longer test case, the difference between capturing the whole screen and only parts of it is very significant. For example, a test case was recorded twice, once using the whole screen and once using half of the screen. The full-screen recording was 41MB and the recording that used half the space was only 18MB. By default, Selenium WebDriver, for example, only takes a screenshot of the website (without the browser's toolbar and without the operating system's UI element).

Previous implementation

There is no previous implementation of this requirement.

Current implementation

The design chosen for this requirement is a screenshot window. In Windows 10, if a user press windows + shift + s key, a dimmed window would

be shown, where the user can draw a square to select an area where a screenshot would be taken. Although this design is not as accurate as giving the exact coordinate of the picture, it is very simple to use and similar concepts are used across multiple software. This design abides with the design consideration because it provides familiarity, enough freedom and constraints. Users are constrained from giving false values (e.g. values that do not represent the recording area).

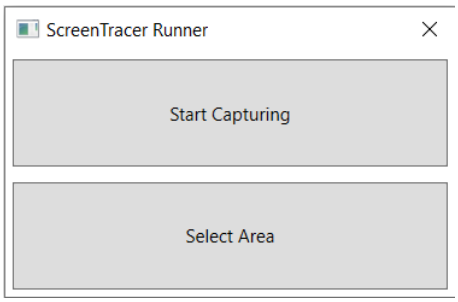

Figure 4.3: Updated ScreenTracer runner

To let users define the recording area, a new button is added to the ScreenTracerRunner window. This button is the bottom button shown in figure 4.3. Clicking this button opens another window. This window starts in a fullscreen mode, has a dimmed background and does not have an exit button. Users have to draw a rectangle by clicking and dragging their mouse. This rectangle represents the area that will be captured by ScreenTracer. After drawing the rectangle, the window will automatically close and the values of the said rectangle will be used as parameters for the screen recorder.

In the back end, this is possible by using the standard C# library. To capture a picture, the function needs four parameters. The x and y value, and the height and the width of the picture, X and Y represent a coordinate where the start point of the picture is. Then the width would extend the x-axis and the height would extend the y axis. This would produce a rectangular area.

These values would be saved into the frames that are then compiled into a ".stc" file. When it is decompiled, the x and y values had to be offset. This is because they moved the pictures from the centre of the screen and this is not desired.

4.4 R3: Viewing standard output

Motivation

According to Zimmerman et al. [12], next to "Steps to reproduce", "Stack traces" is rated as the second most important information in a bug report.

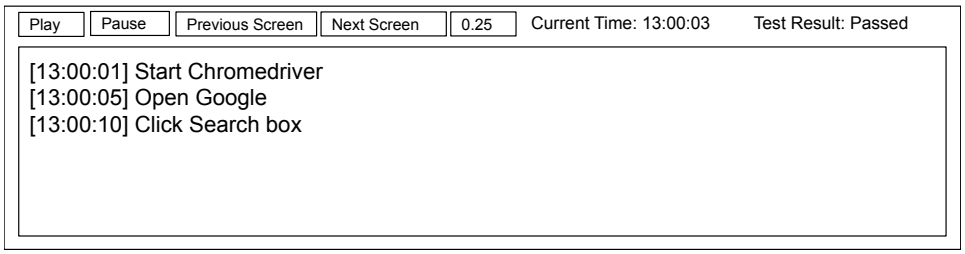

Figure 4.4: Mock-up console output in ScreenTracer

These stack traces are derived from exceptions that are caught during a test. Exceptions are thrown when unprecedented situations happen during a test. For example, during a test to click a button, an exception could be thrown if the button is blocked by another web element. Stack traces includes important information such as which exception is thrown and where it is thrown from. This helps testers to identify the problems faster and navigate directly to where the problem failed in the test source code.

Previous implementation

There is no previous implementation of this requirement.

Current implementation

During the design consideration, there are multiple ideas that are considered. These are:

1. Let the user add comments manually later by editing the video
2. Users can add comments by adding a special line in their test source code
3. Display the standard console output of the test

Options (1) and (2) are eliminated after further considerations because they would increase the number of work that has to be done by the tester. Option (3) is the best because not only it does not require further work, the standard console output is also used widely in programming to display important information. It also records stack traces in case of a failed test case. This allows users to directly lookup for the point of failure in the source code.

A scrolling text box is used to display this information as shown in figure 4.4. A timestamp is added to the console output to allow the user to know which part of the video correlates to the comments. This timestamp is also used to identify when a console output should be displayed. To

simulate a test environment, console outputs are shown gradually as the video progresses.

This design is chosen because it introduces familiarity. It emulates development tools such as Visual Studio and IntelliJ. This provides enough constraint because, unlike the other design considerations, the outputs that are produced here are strictly from the test case. There is no alteration of the output between the test and the playback. This also provides freedom for testers, because they can add lines in the code to print important information and these would be captured in ScreenTracer.

For the back end implementation, it is important to know that ScreenTracer starts the test process. This allows ScreenTracer to redirects the console output from the test. ScreenTracer was implemented synchronously and redirecting the console output would cause deadlock, because ScreenTracer waits until the test ends and the test cannot be ended properly. To fix this problem, the implementation was changed to asynchronous. Instead of waiting for the process to end, ScreenTracer would let the test process run and let it trigger an event handler when it closes. This event handler would save the recording and show the ScreenTracer UI to allow further recording sessions.

To save the console output, a list is used. When the test process sends a console output, another event handler that saves the console output together with the timestamp into the list will be triggered. This list of console output will be saved with the frames and compiled into a ".stc" data format.

This will be decompiled later when the ".stc" file is opened and converted into a video. During video playback, ScreenTracer updates its time value. When this time value updates, ScreenTracer will compare the list of console output with the current video time. If the current time is later than the time when the standard output is taken, ScreenTracer will remove it from the list and display it on the text box.

4.5 R4: Highlighting the web elements

Motivation

Viewing an automated test, without a recording, can be done in two ways. These are either to (1) run the test and observe it or (2) save screenshots on certain parts of the tests and navigate using these screenshots. Both of these however do not provide the information on which elements are clicked or interacted with during each step.

In some cases, web elements fade out when they are clicked or make themselves clear when they are interacted with. For example, when a pop up is closed it will disappear from the screen. In other cases, element interactions are not directly identifiable and these interacted elements are identifiable

only if access to the source code is provided, or if the observer knows what the test is supposed to do.

Previous implementation

There is no previous implementation of this requirement.

Current implementation

To highlight the web elements, a thick red border is chosen. According to Kuniecki et. al. [4], red shows that something is of importance and it commands attention. The colour itself provides intuitive design because it aligns with innate human biology. The highlighting is only added to limited interactions, these are clicks and sending keys. These are the most used interactions and other actions such as hovering over a web element are not highlighted.

To implement the highlighting, restrictive aspects or constraints are done by not allowing testers to directly dictate what should be highlighted and what should not. There is a clear boundary on what the highlight means. Highlighting happens automatically on all elements that are clicked or sent keys. This function is optional so testers have the freedom to either enable or disable this function.

From the user point of view, there are steps needed to use this function and this includes:

1. Developers download the DLL file(s) needed
2. Developers load the DLL file into the project
3. Add the initialization method before the test starts

Currently, it is not possible to simply enable or disable this function through the Selenium GUI and there are reasons why this seemingly complicated approach is chosen as the optimal solution.

To highlight these elements there are two different approaches considered. The first approach would be identifying these elements from the video using video technology. Because ScreenTracer frames are only captured when changes on the screen are made, adding highlighting to these frames could show which areas are currently being worked on. However, this would not function properly because there were issues with the accuracy.

Since ScreenTracer captures areas, the web elements that should be highlighted (from now on these will be called "interest web elements") are not shown clearly. Because the captured area could be either bigger or even smaller than the web elements and this causes inaccurate screenshots. Moreover, when keyframes are taken, the whole recording area is captured and highlighting this would not add any benefits at all. If there are any web

elements interacted with when a keyframe is being taken, this interaction could not be captured.

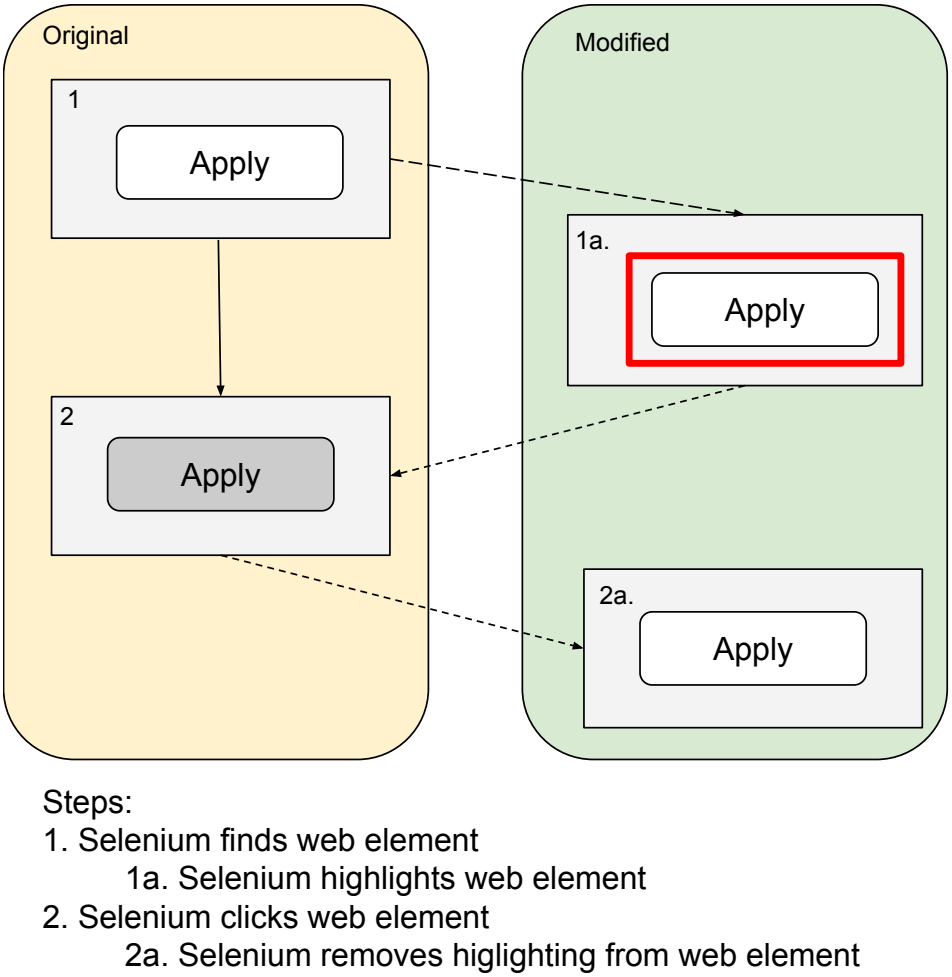

Figure 4.5: Highlighting steps

The chosen approach concerns itself with modifying built-in Selenium functions to highlight the web elements. Figure 4.5 shows the comparison between the original method that selenium provides and the modified method. To click an element, selenium first "find" this element by navigating the DOM (Document Object Model) tree of the website. Then, it represents these elements (buttons, text boxes, icons, etc.) as objects.

These objects are presented in an object-oriented programming context and this means each object has a method they can call. Selenium uses methods such as "Click()" and "SendKeys()" to interact with web elements. If further functionality to highlight and unhighlight web elements is added into these Selenium methods, there would be no accuracy problem.

To implement this, there are several methods that are taken into consideration. These methods can be divided to two parts, which are *static* and *dynamic*. To further understand this, it is important to clarify that for every automated test case written in Selenium, there is a source code for this test. To make an automated test, a source code needs to be first written and then compiled into executable files, and lastly run.

Static method is a method that changes these built-in selenium methods before it is compiled. It modifies the source code, then compiles it. This modification of the source code is done to change the Selenium methods to add the highlighting function.

Dynamic method does not change the source code at all. This method alters the functionality of the original Selenium methods during runtime. This means the modification happens when the test is running.

The current implementation uses both methods because implementing either a pure static solution or a pure dynamic solution proved to be problematic. The issues and the reasoning are discussed thoroughly below.

4.5.1 Static method

The static method revolves around modifying the source code to add the highlighting functions to every selenium interaction. The first step is to identify all interactions in the source code. Pattern matching (such as regex) could be used to do this. RASCAL MPL[1] is a programming language that specializes in transforming programs. This language would enable writing a program that modifies the source code. By using a script, it is possible to automate this modification on every test before it is run by ScreenTracer.

This method was not chosen because it introduces a number of new problems.

1. The pattern matching might not be accurate

   The selenium methods that are used to interact with web elements are mainly "Click()" and "SendKeys()". Using pattern matching would mean knowing what the names of these methods exactly are. If the source code does not use these methods directly but instead uses another method that uses these, pattern matching would not work. For example, if the source code implements a new method called "DoClick()" that adds logging functionality to the click function, and uses this method instead of the Click() method. In this case, instead of identifying every "Click()", it would be preferable to find where the "DoClick()" method is implemented and modify its code. In the worst-case scenario, the "DoClick()" method is imported from an external library and it would not be possible to modify its source code.

[1] https://www.rascal-mpl.org/

2. Providing extra memory

   Modifying the source code can add unwanted functions. It is important to be able to revert the changes that are made. To enable this enough memory has to be provided for the modified code. Following problem (1), it can be concluded that analysing a single class would not be enough. The whole project should be analysed and saved in extra memory space to not overwrite the old project.

3. Identifying the parameter

   The parameter here means the parameter required for the method to highlight the web elements. To highlight the interest element, the element itself is needed to be passed into the method as a parameter. On top of identifying where the interactions are done, the parameter names have to be identified. How to find these heavily depends on the writing style of the code. For example, if the web element object comes from another class, then it would be impossible to find the parameter using pattern matching.

4. Code might not compile

   Modifying the source code could result in a code that cannot be compiled. This adds another problem to an already problem-ridden solution.

After considering these points, this method is not taken as the solution.

4.5.2 Dynamic method

The dynamic method means changing the runtime behaviour of the program instead of modifying the source code. This means no extra memory space is needed and the program is already compiled and running before doing further modifications.

Harmony

Harmony [7] is a framework that provides a way to alter functionality in C# applications by monkey patching its methods during runtime. It does so by redirecting the pointer that points to a method to the modified method. This solves problem (1) because the pattern matching needed is only to find the interaction methods that are provided by selenium. This means the patch has to be specialized to certain versions of selenium since different versions of selenium use different methods. The current patch is suitable for selenium 4.0.

Figure 4.6 is taken from the Harmony website and it shows the concept of monkey patching. It redirects the original to the dynamic method

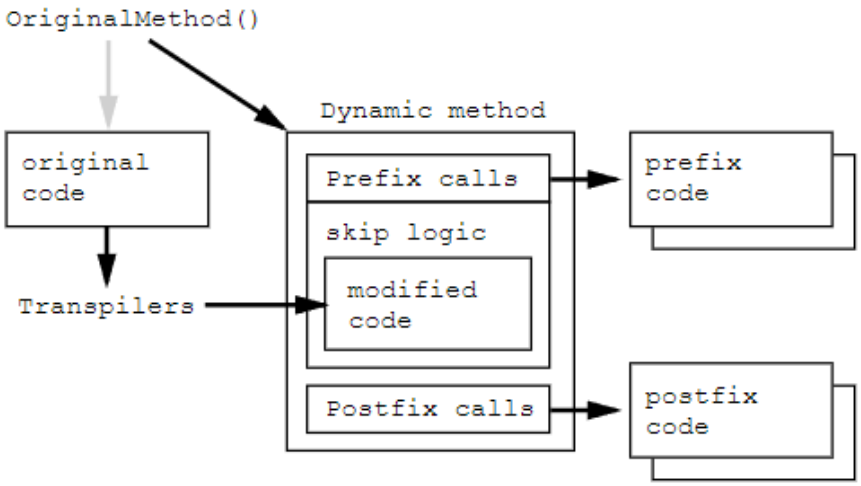

Figure 4.6: The concept of monkey patching, taken from Harmony [7] website

and adds both "Prefix calls" and "Postfix calls" to the original method. Transpilers can edit the code statically, however they are not relevant in this implementation.

Prefix is used to edit the parameter of a method before giving it to the original method. The postfix is used to alternate the return value of the original method. This concept is applied to all interaction methods (clicks and send keys) that selenium offers. Before it interacts with an element, the modified method will highlight it and pause briefly. After the interaction is done, if the element still exists, the highlighting will be undone. In some cases, the elements are gone after an interaction. E.g, clicking a link that moves the website to another page.

These patches have to be applied to multiple test cases. This code is reusable and there is a method to distribute reusable code easily. Compiling patches into DLL Files is the next step in implementing this requirement.

DLL File

Downloadable library or DLL for short is compiled code that can be used by another application. In this context, the DLL files consist of the harmony patches and the initialization code. To make these accessible throughout multiple test codes, these lines of code would be compiled into DLL files. This serves the purpose to (1) enable code injection, (2) introduce reusable codes and (3) easily updatable patches. This serves as a solution for developers that want to enable this highlighting function into their test codes since loading and calling codes from DLL file(s) is industry-standard.

Writing DLL file(s) using C# is different than using C++. In C++

there are functionalities that are innate to it that supports exporting static methods and the lack of this functionality in C# would later prove to be problematic. To compensate for this, a NuGet package called DLLExport is used. This package enables developers to export static methods from DLL file(s) that are written in C#. Exporting the methods means that the pointers to these methods are readily available (the pointers are listed in the pointer table). Not exporting the methods in itself should not be problematic, since they can still be resolved by the IDE(Integrated Development Environment), however, this proves to be another matter when using code injection.

Code injection

The patches are written as .NET[2] DLL and to activate it, it first has to be loaded into the test code and initialized. Importing and initializing the patches require lines of code to be added to the original test code. Because the dynamic method refrains from any form of modification of the source code, another method to do this process without modifying the source code is needed. After further research, a hacking method called code injection could provide a great solution to this problem. Code injection is a type of attack to inject lines of code to be executed by an application. Implementing this enables ScreenTracer to inject the test program with codes that load and run the harmony patches during its runtime.

Figure 4.7 describes the final concept of code injection. After the user starts to record a session using ScreenTracer, it would use code injection to automate the steps necessary to activate the harmony patch.

The steps taken to do code injection, inspired from the website Codeporject[3], are:

1. Starts the test process
2. Get the PID(Process ID) of the process
3. Create a thread to allocate enough memory for the path to the patch
4. Load the path into this allocated memory
5. Inject the code to call "Load library" method
6. Inject code to call the initialization method from this library

Step (1) is implemented by running the executable file of the test code and step (2) is implemented by calling a get method onto the process object

[2]https://dotnet.microsoft.com/en-us/

[3]https://www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Proces

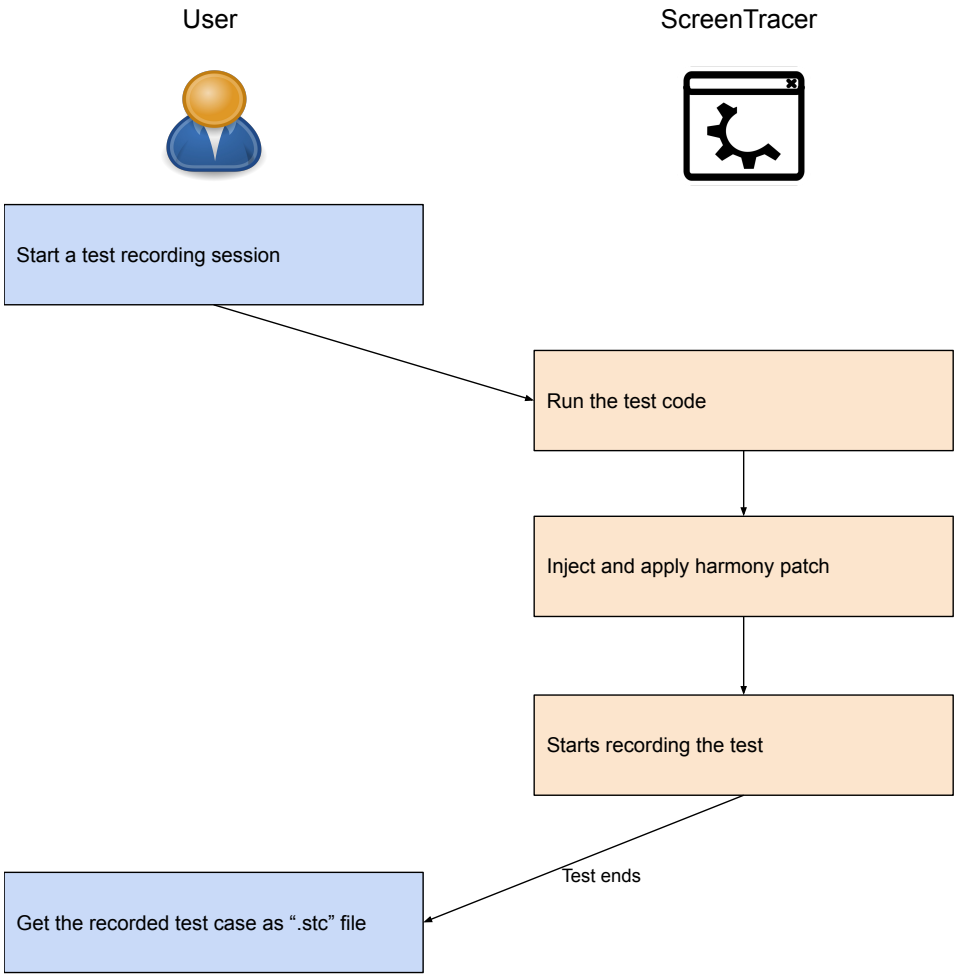

Figure 4.7: Concept of code injection

created in step (1). Steps (3) and (4) don't cause problems and were successful. During the implementation of step (5), there were issues found that would eliminate this solution as a candidate. The first major issue stemmed from the compatibility issue of the downloadable library (DLL) file with the test code. .NET Projects can be created in either .NET Framework or .NET core and depending on the project type, a suitable DLL file that is created using a compatible framework version would need to be written. For example, a test code written using .NET Framework could not load a DLL file written in .NET core using the code injection method. This problem was very hard to debug because there are no debug messages shown when implementing code injection, and versions would be redeemed as incompatible if step (5) caused the test code to crash. The trial and error method was used to find the compatible versions and this meant that

for a single test code, multiple versions of the same DLL files were required. After trying multiple versions, the development was constrained to test codes that are written in .NET Core (5.0) and DLL files that are written in .NET Framework (4.7.2).

After constraining the development environment, step (6) would produce another issue. As mentioned in the previous section (DLL file), the initialization methods were not found and trying to call this method caused the program to crash. DLLExport fixed this issue, but the DLL file(s) created by DLLExport could not find its dependencies. In this case, the method that has to be called is inside the "InjectMe.dll" file and this file depends on the harmony DLL file and the Selenium WebDriver DLL file. When using DLLWalker it could be checked that these DLL files are not missing dependencies, but during code injection, this lack of dependency caused runtime failures. A solution to load the dependencies onto the test code before compile time has been tried and it could not fix the problem.

At this point, it was best to reconsider this approach because of multiple reasons. The time constraint was not enough and this approach introduces constraints that are not ideal to have (for example the framework versions). Implementing code injection would only save developers a few steps which are (1) loading the DLL file(s) and (2) adding the initialization code into the start of the test code. It is not productive to only save two simple steps while at the same time introducing multiple points of failure.

4.6 Summary

The functionalities that were planned to be added and their final stand are described in table 4.1. All of the main requirements are functional, they have been tested multiple times and works reliably. The only optional requirement that was added was the GUI Remake, while the other requirements were not added.

There are efforts made to implement **OR8**, but it had to be cancelled because implementing this would require a complete remake of ScreenTracer. This would take too much time and would derive the main goal of this work. The other optional requirements are dependent on this requirement. Even though it is possible to implement other requirements before implementing MVVM, it would require tremendous effort to redesign them later into MVVM.

ID	Requirement	Importance	Status
R1	When a video file is open in ScreenTracerViewer, the user can change the video playback speed.	High	F
R2	Before starting a recording session, the user can select the recording area.	High	F
R3	When a video file is open in ScreenTracerViewer, the user can view the standard output from the recorded test.	High	F
R4	During recording, ScreenTracer highlights GUI Elements that the Selenium WebDriver are interacting with.	High	F
OR1	Users are able to cancel a recording session.	Low	NF
OR2	ScreenTracer does not record before the Chrome Browser opens and after it closes.	Low	NF
OR3	ScreenTracerViewer GUI Redesign to implement new functionalities.	High	F
OR4	The selected area is saved as preset.	Low	NF
OR5	The next screen and previous screen function of ScreenTracerViewer must load under 1 second.	Low	NF
OR6	ScreenTracerViewer has a control panel in full screen mode.	Low	NF
OR7	ScreenTracer can select the project to record from the GUI.	Medium	NF
OR8	ScreenTracerViewer is implemented using the MVVM Model for future development.	Medium	NF
OR9	The previous and next screen button can be held down to skip multiple screens.	Low	NF

Status

F = Finished

NF = Not finished

Table 4.1: Requirements table

Chapter 5

User Study

After the implementation phase, a user study is conducted to measure the effectiveness of ScreenTracer in the industry. The user study is designed using the GQM(Goal/Question/Metric) method. According to Wohlin et al. [11], to measure in a purposeful way, it has to specify the goals, trace those goals to the data, and interpret the data with respect to the stated goals. These are defined as:

1. Conceptual level **(Goal)**. A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. Objects of measurement are products, processes, and resources.

2. Operational level **(Question)**. A set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterization model. Questions try to characterize the objects of measurement (product, process and resource) with respect to a selected quality aspect and to determine its quality from the selected viewpoint.

3. Quantitative level **(Metric)**. A set of data is associated with every question in order to answer it in a quantitative way (either objectively or subjectively).

In this chapter, the goal, question, metric of this user study are elaborated further and presented in a structure that is inspired by Jedlitschka et al. [3].

5.1 Concept

This section underlies the concepts for this user study.

5.1.1 Goal Definition

This study is motivated by a need to research the impact of ScreenTracer on debugging. As explained in chapter 2, there are advantages that a video form debug report has in comparison to static images. These benefits could affect the speed of debuggers to identify bugs. The goal is to check if ScreenTracer helps its users find these bugs faster.

Purpose

The purpose is to evaluate the difference in speed to find bugs when ScreenTracer is involved. The user study compares the performance of two groups. The first group tries to find bugs with similar resources to what it is given in the industry. The second group has the advantage of being able to use ScreenTracer on top of having all resources provided for the first group. This comparison provides insight into if ScreenTracer provides a speed advantage to its users.

Quality focus

The main effect studied in the user study is the effect ScreenTracer has on debugging speed. The difference of time between subjects that use ScreenTracer to find bugs will be compared to subjects that don't have ScreenTracer to find the same bug. The metric used for this is second(s).

Context

The user study is run within the context of ScreenTracer and the industry. Although it is speculated that ScreenTracer could help its users find bugs faster, there is no previous study done to prove this. To make a fair comparison, groups that do not have ScreenTracer have a debugging experience that is similar to the environment in the industry.

5.2 Planning

This section explains the planning for this user study.

5.2.1 Hypotheses formulation

Hypotheses are important in experiments because they state clearly and formally what are going to be evaluated from an experiment. In this study there is only one hypothesis, which is, informally:

- ScreenTracer users can find bugs faster. Most (90%) of the participants are test automation engineers that are used to getting bug reports in

the forms of static images with brief descriptions of said images. This means it is expected that they perform better when provided with ScreenTracer.

From this statement, a formal hypothesis can be derived.

- Null Hypothesis, H_0 There are no speed difference between ScreenTracer users and non-users in finding bugs.

This means these data need to be collected:

- Speed to find bugs using ScreenTracer, measured in seconds
- Speed to find bugs without using ScreenTracer, measured in seconds

5.2.2 Variables selection

The independent variables are experience in Selenium and individual skill level. The dependent variable is the time taken to find bugs.

5.2.3 Selection of Subjects

The subjects are chosen based on their experience, especially with quality assurance. Most of the subjects (80%) are software engineers that work with Selenium WebDriver daily to create test automation software. Out of these subjects, 5 of them are working students and 3 of them are employees. One subject only has basic programming understanding, but as an employee and used to working in an industrial environment. One subject did not have any experience with Selenium.

5.2.4 Threats to validity

An important question to ask to every experiment is how valid the results can be. According to Wohlin et al. [11], there are four types of threats to validity. These can be categorized into *conclusion*, *construction*, *internal* and *external validity*.

Conclusion validity is concerned with the relationship between the treatment and the outcome. There should be a statistical relationship with a given significance. In the scope of this study, there were ten participants. If there are data that has to be taken out of the sample, this significantly reduces the validity of this study, which already has a low number of participants. To reduce this effect, variation is introduced into the user study.

Internal validity makes sure that if the relationship between the treatment and the outcome is observed, it must be a causal relationship and it is not a result of a factor that cannot be controlled. The factors that cannot be

controlled here are the prior knowledge and experience of each participant. To counter this problem, most of the participant (80%) chosen actively works as test automation engineer and have a basic understanding of the tools used in this study. This means the effect of the independent variable would be minimized.

Construct validity is concerned with the relation between theory and observation. If the relationship between cause and effect is causal, two things must be ensured: (1) that the treatment reflects the construct and (2) that the outcome reflects the construct of the effect well. This raises concern towards the method of the user study. The user study would provide bugs for subjects to find. If these bugs are not complex enough, the user study might not reflect the construct well. To reduce this effect, the bugs provided, or the problems are presented in an ascending order based on their difficulty. This means they start with the easiest problem and ends with the hardest problem.

External validity is concerned with generalization. It is important to know if the results of the study could be generalized outside the scope of the user study. This validity issue is countered by the design of the user study, which emulates the condition of the industry.

5.3 Operation

This section elaborates on how the user study is laid out and executed.

5.3.1 Preparation

The preparation phase is used to prepare the bugs that have to be found by the subjects. The bugs are found in SynchroPC[1], a software that provides a platform to create appointments to discuss scientific papers with colleagues. Through these bugs, failed test cases are created to simulate the case of failed tests in the industry. These test cases are attached in appendix A. This bug finding activity will also be referred to as problems, as in problems in a test.

Steps that happen in the industry are; (1) Testers write the test-automation program (or test cases) and (2) run said program to test software, which is SynchroPC in this context. (3) After the test program run, debug reports are produced automatically that provides information about the test. Using this debug report, testers can (4) trace where the bug comes from and report it to the software developer. This debug report usually has screenshots and brief descriptions of what the program was doing as the screenshot was taken. In this step, the comparison is made to check if ScreenTracer would prove to be advantageous for its users.

[1] https://synchropc.se.uni-hannover.de/login/

The screenshots are to simulate the debug reports that testers get in the industry. There are two different types of screenshots, either they are taken after every major step (descriptive screenshots) or taken after every interaction (interactive screenshots). Descriptive screenshots have a description on them and they also have information about the exception thrown in cases of failure. Interactive screenshots are named after the time they are taken in so testers can see which ones come first. To differentiate these screenshots, they are saved in different folders.

This means there are two variations of bug reports, static images and video files. In this study the first three steps are "done" and the subjects are given different resources to find the bugs depending on which variation of the study they get. Both variations provide access to the source code of the automated test, SynchroPC and screenshots of the test. Both the source code and SynchroPC are needed by testers to do steps (1) to (3). The difference is that one variation has access to ScreenTracer and the recorded video of the test run in step (2) on top of every resource provided to other variations.

It is not enough to divide the participants into two groups and let half of the subjects solve all problems with one variation, and the other half solve all subjects with the other. There are also factors that need to be accounted for, such as:

- Software developers perform differently.
- A bug can only be found once, this means direct comparison is not possible.
- The difficulty of finding bugs cannot be objectively measured.
- Subjects "warm-up", meaning they perform better the later the test goes.

As explained in the construct validity, problems are presented in gradually increasing difficulty. To further clarify this, these bug-finding problems are made by doing these steps: (1) Find a bug in SynchroPC, (2) Construct a coherent test case that could be understood, and make this fail and (3) record this failed test case and use this as the assignment for the subjects. Subjects have to answer the question *"What is the bug in this test case"*. These test cases are attached in appendix A.

This adds to the fact that there is a conclusion concern in this study because of the lack of participants. To solve this problem, more variance has to be introduced.

Table 5.1 shows some possible combinations of how the user study is done. Because there are three bugs and two ways to find each bug, the number of possible combinations is 2 * 2 * 2 = 8. In the final study, the final variations

	Bug A	Bug B	Bug C
Participant A	S	H	S
Participant B	S	S	H
Participant C	H	S	S

Table 5.1: User study model

chosen were: "S,H,S" , "S,S,H" , "S,H,H" , "H,S,S" , "H,S,H". S here represents "with ScreenTracer" and "H" represents "without ScreenTracer" or "with screenshots (static images). The variations "S,S,S" , "H,H,H" and "H,H,S" are taken out because there were only ten participants. Limiting the number of variations to five enabled direct comparison between two variations.

For the first bug, they were given ten minutes and they were given fifteen minutes for the other two bugs. If a bug could not be found within the time limit, the time would be recorded as (limited time + (one minute)). The time limit is set based on the difficulty of each bug. The first bug is thought to be easier to find than the other two. The order chosen was from the easiest bug to the hardest bug, which help the subjects "warm-up" on the easiest bug first to be ready to solve the incrementally harder tasks. This can be seen in table 5.2.

To make the user study structured, it is divided into three phases, which are the tutorial phase, the test phase and the interview phase.

Tutorial Phase

The tutorial phase explains the subjects about the relevant information to the testing phase. The subjects need basic levels of understanding about what SynchroPC does and how does it works before going on to the next phase. They also need to know what is ScreenTracer, how it works, and what they are supposed to do. The tutorial phase is documented so it can be conducted in a consistent manner.

Test phase

In this phase, the recording is started and subjects had to solve all the problems given. If the subject asks a question, it should follow into one of these categories to be answered:

1. Questions about the functionality of SynchroPC
2. Question about the functionality of ScreenTracer
3. Questions about the functionality of Visual Studio

4. Questions about the user study

These questions do not directly help the subject in solving the problem, and they only help to let the user focus more on the debugging aspect of the study. After every problem, the solution is presented to the subject.

Interview phase

After the test phase, an interview is conducted to gain feedback from the subjects. The questions revolved around the new functionality of ScreenTracer and if they think that ScreenTracer could be beneficial in the industry.

5.3.2 Execution

This section explains how each user study is executed. There are three phases in the user study, which are the tutorial phase, the test phase and the interview phase which combines sixty minutes for each user study session. Sessions are held online through screen sharing using Zoom. Before the user study, subjects have to sign a consent form that is attached in A.

The user study was executed over three weeks, during which ten sessions took place. For the first six sessions, the tutorial phase only took five minutes. The first three subjects did not have any problem with this, but two subjects said that the tutorial phase was not comprehensive enough and it affected their performance in the test. After reconsideration, this time limit was taken out, and subjects are encouraged to ask questions.

5.4 Evaluation

In this section, the result of the user study is presented in measurable data and are described.

5.4.1 Data Reduction

In the user study, the result of subject number nine had to be taken out because said subject lacked the prior experience of Selenium in comparison to other subjects. This would make an unfair comparison. The first result of subject number one had to be taken out because of technical issues during the interview. This can be seen in table 5.2.

5.4.2 Data analysis

After interviewing all participants, the data taken is listed in a table. Table 5.2 shows the time needed by a participant to solve a bug. The numbers directly below the Time(s) cell represents the three different bugs. The

ID	Variant	Time(s)		
		A	B	C
1	S,H,S	660 (*)	475	438
2	S,S,H	58	599	307
3	S,H,H	660	960	628
4	H,S,S	80	727	514
5	H,S,H	385	960	524
6	S,H,S	660	960	380
7	S,S,H	660	960	394
8	S,H,H	660	664	700
9*	H,S,S	660	660	960
10	H,S,S	660	960	640

*	Data needs to be taken out of sample
	Bug not found

Table 5.2: Result of user study, refer to 5.4.1 and 5.3.1

variant shows the variant that each subject got, and this is read from left to right. For example, the first row means subject one solved bug A with variant S in 660 seconds, bug B with variant H in 475 seconds and bug C with variant S in 438 seconds.

5.4.3 Hypotheses testing

To check the hypotheses H_0, the average time needs to be measured.

Bug	Bug A		Bug B		Bug C	
Variant	H	S	H	S	H	S
Average time taken	375	539.6	764.75	841.2	539.75	493

Table 5.3: Average time taken, calculated after data reduction

Table 5.3 shows the average time taken to find different bugs using different variations. On the first and second bug, subjects with the H variant could find the bugs faster. But on the third bug, subjects with the S variant could find the bug faster.

H_0. There is no significant difference in speed between participants that use ScreenTracer and those who don't. In regard to this hypothesis, it cannot be rejected.

5.5 Discussion

The goal is to prove that ScreenTracer helps subjects find the bugs faster using data that is evaluated in section 5.4. After conducting the user study, there are multiple reasons found on why the data taken could not prove the goal. These reasons are as followed:

- Provided bugs for the problems are not ideal
- High individual variance
- Problems could not be solved
- Not enough time

The bugs that are used as problems in this user study did not really favour ScreenTracer. To provide the bugs as problems for the user study, it has to be found first by using and testing the buggy software, which is SynchroPC. The user study simulates the condition in the industry and that is why these found bugs have to be represented as a test case in a coherent manner. Each test case has a purpose to test a certain functionality aspect. Test cases also have a "should-value" and an "is-value", but these are hidden from the subjects because the bugs could be directly found if these values were given, and there would be no test at all. These factors and the added time constraint resulted in test cases that are very similar to each other.

All test cases had the bugs found at the end of them. And even though it is possible to extend a test case and modify it to make the bug found in the middle of the test case, it would take away the goal and would not fit into the time constraint. For example, if a bug in figure2.2 exists, the test case can be made to test the search box function, or to test another function that has to use this search box function beforehand. This study did not introduce any intermediate error that is discussed in 2.3. The lack of variety in all the bugs found also contributed to the next problem.

Subjects for this study, which are all software engineers, perform differently. This also applies in debugging, but because of the lack of variance in problems, there is one best approach to solve them the fastest way. This approach is to read the console output or to read the exception given, and then directly search the bug inside the source code around the problem area. This means subjects that uses this algorithm to solve each bug-finding problem performed the best, and other approaches performed worse to the point that the problems could not be solved inside the time constraint. The subjects that perform the worst in the test tried to understand what the test is supposed to do first, this is a viable approach in the industry, but not a great approach if a time limit is given.

This caused a lot of subjects to not be able to solve the first two problems but solve the third problem. This is because they understood that there is a

certain method to solve the problems and after doing the first two problems, they started to recognize the pattern. This resulted in the lack of viable data for the first two bugs. In table 5.2 it could be seen that only three subjects solved problem A and only four subjects solved problem B. Although the time is measured with mock data, there are too much mock data to be able to prove the hypothesis.

Another problem is also the limited time for the tutorial phase. In the tutorial phase subjects had to absorb a lot of information. This includes information about SynchroPC, ScreenTracer, and the problems that they had to solve. To satisfy the time constraint, only the basic information was explained for SynchroPC. A subject that performed worse when using ScreenTracer said, that he would much rather prefer ScreenTracer but the lack of understanding about SynchroPC made him perform worse than usual. Another subject understood SynchroPC despite the very basic explanation and solved problem A in under one minute.

Chapter 6

Conclusion and future work

This chapter has 2 subsections, which are the conclusion and future work. The conclusion briefly reminds the main goal of this work and explains the end state of the goal. This includes goals set for the development part of the work and the user study.

Future work discusses the possibility in the future to make ScreenTracer a more complete software for the industry. The feedback given by subjects during the user study is considered to add more functionality to the software. This sub-chapter also explores the possible improvements that can be done to the user study, in case another user study is conducted in the future.

6.1 Conclusion

The main goal is to implement more functionality for ScreenTracer and to test if these functionalities would help testers find bugs faster. The first part of this work is the development phase, in which all main requirements are implemented. Implemented and unimplemented requirements can be seen in chapter 4.

After the development phase, the user study was planned and designed. The user study had ten participants that are all experienced with quality assurance. In the experiment, they had to find three bugs and the time taken to find each bug are measured. The final verdict is that the data were not sufficient to reject the null hypotheses. This was caused because of the lack of sufficient material (bugs found in SynchroPC), the high individual variance of each subject, the lack of the number of participants and the lack of time.

During the interview phase, most feedback given towards the functionalities added were positive. These functionalities were thought to be very important, and some subjects stated explicitly that they would prefer to use ScreenTracer rather than other bug reports. Some subjects gave feedback on how ScreenTracer can be improved in the future and this is explored in

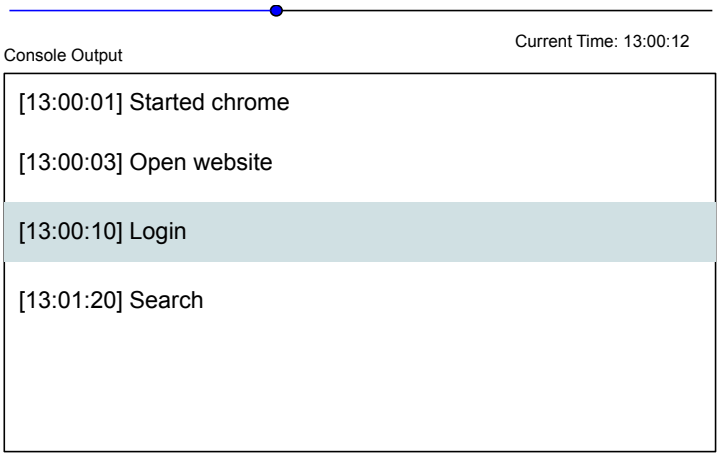

Figure 6.1: Updated console output concept

the next subsection.

6.2 Future Work

6.2.1 ScreenTracer

To make ScreenTracer more suitable for the industry, multiple improvements can be done. This can be done with either more functionality or more reliability. From the interview phase of the user study, half of the subjects wanted more options to navigate the video. This means instead of just scrolling the time slider, it would be very helpful if the console output texts are clickable and clicking them would change the video to the time when this text was produced.

Figure 6.1 demonstrates what the updated console output would look like. In this figure, it could be seen that the current time is 13:00:12 and ScreenTracer highlights the output that is relevant to the current frame. Another function is when a comment is clicked, ScreenTracer would change the current time to the time stated on the left side of the comment. This method of navigating through the frames allow users to jump into the most relevant information that they want to see.

Other functionality features that could be added into ScreenTracer has more to do with the back-end. ScreenTracer can only start tests locally, which hinders methods of running test automation, for example using Docker[1], or running it remotely in a server. Running a test locally means during the test, users should not interact with the machine running the test, because this causes interference in the recording session. Allowing tests to

[1]https://www.docker.com/

run in containers or remotely mitigates this problem. If this is implemented, the use of Jenkins[2] to schedule automated tests would be possible.

This could be supported by allowing ScreenTracer to start recording not only from the GUI but also by providing a CLI(Command Line Interface) for users. Since most interactions with servers are faster by using CLI rather than GUI.

Another major point is the lack of support for NuGet packages such as NUnit[3]. Although it is currently possible to run applications written in NUnit using ScreenTracer, R3 and R4 do not support this framework. This is because NUnit starts its threads and ScreenTracer needs access to a thread directly to implement R3 and R4. NUnit is a major testing framework and it would benefit ScreenTracer greatly to support these packages.

6.2.2 User Study

If the user study is to be restarted with a bigger time budget, multiple points can be improved on. The major setback was the software chosen for this study. SynchroPC itself is not a complicated program and in turn, also provide simple bugs and this restricted the study from introducing more complicated problems. All the problems provided to the subjects were very similar and did not introduce intermediate error explained in 2.3. The suggestion would be to provide more complex problems.

Another problem is that subjects were not familiar with SynchroPC. This proved to be detrimental to the study because the "warming up" effect is significantly higher in this study as predicted. To fix this problem the software could be changed to software that is regularly used by the subjects. Another solution would be to send a tutorial video to all subjects before the test and they have to watch this before the test. This would save time from each session and allow subjects to review the tutorial given more than one time.

[2]https://www.jenkins.io/
[3]https://nunit.org/

Appendix A

Appendix A

Files relevant to the user study

CONSENT FORM

Investigation of the effectiveness of ScreenTracer application in finding bugs

Description

My name is Malvin Andika Tandun and I am conducting this study for my bachelor thesis. This study investigates the effectiveness of the ScreenTracer application in helping the participants to find bugs. In this experiment, participants are given 3 bug reports. There are two types of bug reports, the first type is a video bug report and the second type is screenshots. The video bug report will be viewed using ScreenTracer. The experiment is divided into 3 phases, the tutorial phase, the testing phase and the interview phase. In the tutorial phase, the interviewer will clarify to the participants the basic functionalities of the software that will be used in the bug report. Then the participants will try to find bugs within the time limit.The first bug report will have a time limit of 10 minutes, the other two 15 minutes. At the end, the interviewer will ask the participants for feedback. The duration of the experiment ranges from 45 to 60 minutes.

Risk and benefits

The participation in the experiment is not associated with any risks or benefits.

Cost and compensation

There are no costs and no compensation.

Confidentiality

All data collected are strictly used for academic research purposes by the software engineering department at Leibniz Universität Hannover. The study will be recorded with a voice recorder and stored for evaluation. Screen sharing will be used but it will not be recorded. During the evaluation, the identity of participants are kept confidential.

Termination of experiment

The participant can stop or end the experiment at any time. The participant also has a right to revoke their consent at any time without repercussions.

Voluntary consent

The points listed above have been explained to me by the presenter. Any questions that I have before, during and after the experiment will be answered by the presenter. By signing this document, I have understood and read the document, and that I voluntarily wish to participate in this experiment.

(First name, Last name)	(Date, place, signature)

Confirmation of the experimenter

I confirm that I have explained all the points listed above and I will answer questions that are related to the experiment.

(First name, Last name)	(Date, place, signature)

Q1: Explain what SynchroPC is.

Setup: SynchroPC is opened and logged in in Google Chrome.

(Show the personal view page of SynchroP) Explain: *SynchroPC is a software used by scientific workers to discuss papers with their colleagues*.
(Click on the Initial Agenda link) and all the available papers will be listed.
(Select the first 3 papers) Select papers and then explain about the time.
(Click on create agenda button) Explain the results
(Go back to personal view) Explain that the results are shown here and the process of accepting a paper
(Accept, reject and suspend a paper)
(Show pc meeting tab) Explain that paper status can be changed
(Change the status of a paper)
(Go back to personal view) Show the changed paper status

Q2: Explain what ScreenTracer does.

Setup: ScreenTracer is open

(Open example test)
(Plays the video) explain what this video is
(Show speed function)
(Show console output)

Test phase
- Test Bug A
- Test Bug B
- Test Bug C

Explain answer after every test

Interview Phase:

Questions:
1. How do you find the software?
2. Can you find bugs quicker?
3. What do you think about the highlighting?
4. What about the speed slider?
5. What about the console output?

```
//Login to synchropc
driver.Navigate().GoToUrl("https://synchropc.se.uni-hannover.de/login/");
seleniumUtil.SendKeys("//input[@name='email']", "malvin.tandun@stud.uni-hannover.de");
seleniumUtil.SendKeys("//input[@name='password']", "synchro123");
seleniumUtil.LogScreenshot("Open synchropc and login");
seleniumUtil.ClickElement("//button[@type='submit']");

seleniumUtil.LogScreenshot("Synchropc opened");

seleniumUtil.LogScreenshot("Select papers");
seleniumUtil.ClickElement("//a[contains(text(),'Initial Agenda')]");

//Select 3 papers
for (int i = 1; i <= 3; i++)
{
    seleniumUtil.ClickElement($"//option[@value='{i}']");
}

//Set the calendar and create the schedule
SetCalendar();
seleniumUtil.LogScreenshot("Calendar set");

//Change back to personal view
seleniumUtil.ClickElement("//a[contains(text(),'Personal View')]");
seleniumUtil.LogScreenshot("Change to personal view");

//Accept all paper
AcceptPapers(3);
seleniumUtil.LogScreenshot("Accepted all papers");


//End the test
driver.Quit();
```

Figure A.1: Problem A source code

In problem A (Figure A.1) the button in SynchroPC main page should not be enabled. They are enabled because of a bug and caused the test to fail.

In problem B (Figure A.2), user should be able to change a paper's status and accept it. This failed, and thus causes the test case to fail.

Problem C (Figure A.3) shows a failure that is not detected by the test case. The test should have created 2 notifications, but only one was created.

```
//Login to synchropc
driver.Navigate().GoToUrl("https://synchropc.se.uni-hannover.de/login/");
seleniumUtil.SendKeys("//input[@name='email']", "malvin.tandun@stud.uni-hannover.de");
seleniumUtil.SendKeys("//input[@name='password']", "synchro123");
seleniumUtil.LogScreenshot("Open synchropc and login");
seleniumUtil.ClickElement("//button[@type='submit']");

seleniumUtil.LogScreenshot("Synchropc opened");

seleniumUtil.LogScreenshot("Select papers");
seleniumUtil.ClickElement("//a[contains(text(),'Initial Agenda')]");

//Select 1 papers
seleniumUtil.ClickElement($"//option[@value='4']");


//Set the calendar and create the schedule
SetCalendar();
seleniumUtil.LogScreenshot("Calendar set");

//Change back to personal view
seleniumUtil.ClickElement("//a[contains(text(),'Personal View')]");
seleniumUtil.LogScreenshot("Change to personal view");

//Click next paper
seleniumUtil.ClickElement("//button[@id='next_paper_button']");
seleniumUtil.LogScreenshot("Clicked next paper");

//Open manage pc meeting in a new tab
seleniumUtil.SendKeys("//a[contains(text(),'Manage PC Meeting')]", Keys.Control + Keys.Enter);

ReadOnlyCollection<string> tabs = driver.WindowHandles;

driver.SwitchTo().Window(tabs[tabs.Count - 1]);

//Go to sessions and change the paper status to preparation
seleniumUtil.ClickElement("//a[contains(text(),'Sessions')]");

seleniumUtil.ClickElement("//th[@class='field-start_time nowrap']//a");

seleniumUtil.ClickElement("//select[@name='status']");
seleniumUtil.ClickElement("//option[@value='SU']");
seleniumUtil.ClickElement("//input[@value='Save']");
seleniumUtil.LogScreenshot("Changed paper status to preparation");

driver.SwitchTo().Window(tabs[0]);

//Try to accept the paper
AcceptPaper();


//If paper not accepted, test fails
if(driver.FindElements(By.XPath("//li[@class='list-group-item agenda-current']")).Count != 0)
{
    throw new Exception("Paper not accepted");
}

//End the test
driver.Quit();
```

Figure A.2: Problem B source code

```
try
{
    //Login to synchropc
    driver.Navigate().GoToUrl("https://synchropc.se.uni-hannover.de/login/");
    seleniumUtil.SendKeys("//input[@name='email']", "malvin.tandun@stud.uni-hannover.de");
    seleniumUtil.SendKeys("//input[@name='password']", "synchro123");
    seleniumUtil.LogScreenshot("Open synchropc and login");
    seleniumUtil.ClickElement("//button[@type='submit']");

    seleniumUtil.LogScreenshot("Synchropc opened");

    seleniumUtil.LogScreenshot("Select papers");
    seleniumUtil.ClickElement("//a[contains(text(),'Initial Agenda')]");

    //Select 3 papers
    for (int i = 1; i <= 3; i++)
    {
        seleniumUtil.ClickElement($"//option[@value='{i}']");
    }

    //Set the calendar and create the schedule
    SetCalendar();
    seleniumUtil.LogScreenshot("Calendar set");

    //Change back to personal view
    seleniumUtil.ClickElement("//a[contains(text(),'Personal View')]");
    seleniumUtil.LogScreenshot("Change to personal view");

    //Accept all paper
    AcceptPapers(3);
    seleniumUtil.LogScreenshot("Accepted all papers");

    //Set a notification
    CreateNotification("notification 1");
    seleniumUtil.ClickElement("//button[@id='next_paper_button']");

    //Set another notification
    CreateNotification("notification 2");
    seleniumUtil.LogScreenshot("Created 2 notifications");

    //End the test
    driver.Quit();

} catch (Exception e)
{
    seleniumUtil.LogScreenshot(e.ToString());

    driver.Quit();
    //throw e;
    Environment.Exit(-1);
}
```

Figure A.3: Problem C source code

Appendix B

Appendix B

Contents of CD

Contents of the CD includes:

1. ScreenTracer source code
2. Transcript of the user study
3. Consent form
4. The bachelor thesis in digital form (.pdf)

Appendix C

Acknowledgements

I would sincerely thank the following people who have helped me in this journey.

M. Sc. Jianwei Shi my supervisor for providing me with feedbacks and keeping me motivated.

Prof. Dr. rer. nat. Kurt Schneider and Dr. rer. nat. Jil Ann-Christin Klünder for examining the thesis.

My friends and family who provide me with encouragement.

My colleagues who provided valuable information and taking part in the user study.

Bibliography

[1] H. Holzmann. *Videounterstützte Ablaufverfolgung von Tests für Anwendungen mit grafischer Benutzeroberfläche.* Bachelor's thesis, Leibniz Universität Hannover, Hannover, August 2011.

[2] J. Hunt. *Agile Software Construction.* Springer, 2006.

[3] A. Jedlitschka and D. Pfahl. Reporting guidelines for controlled experiments in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages pp. 95–104, 2005.

[4] M. Kuniecki, J. Pilarczyk, and S. Wichary. The color red attracts attention in an emotional context. an erp study. *Frontiers in Human Neuroscience*, 9, 2015.

[5] A. M. Memon. GUI testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.

[6] Microsoft. C# Coding Convention. `https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions`. last accessed: 2021.09.30.

[7] A. Pardeike. Harmony. `https://harmony.pardeike.net/index.html`. last accessed: 13.01.2022.

[8] R. Pham, H. Holzmann, K. Schneider, and C. Brüggemann. Beyond plain video recording of gui tests: Linking test case instructions with visual response documentation. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 103–109, 2012.

[9] W. W. Royce. Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON, Los Angeles*, pages 1–9, August 1970. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338.

[10] Techwell and Tricentis. The Evolution of Test Automation. `https://www.tricentis.com/wp-content/uploads/2018/05/Tricentis-Report-The-Evolution-of-Test-Automation-2018.pdf`, 2018. last accessed: 2021.09.30.

[11] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[12] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36:618–643, 09 2010.