



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Automated GUI Testing: A Comparison Study With A Maintenance Focus

Master's thesis in Software Engineering

**PATRIK HAAR
DAVID MICHAËLSSON**

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

MASTER'S THESIS 2018

**Automated GUI Testing:
A Comparison Study With A Maintenance Focus**

PATRIK HAAR
DAVID MICHAËLSSON



Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Automated GUI Testing: A Comparison Study With A Maintenance Focus
PATRIK HAAR
DAVID MICHAËLSSON

© PATRIK HAAR, 2018.
© DAVID MICHAËLSSON, 2018.

Supervisor: Robert Feldt, Computer Science and Engineering
Advisor: Tomas Odin, CANEA Partner Group AB
Examiner: Eric Knauss, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2018

Automated GUI Testing: A Comparison Study With A Maintenance Focus

PATRIK HAAR

DAVID MICHAËLSSON

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

Automated GUI (Graphical User Interface) tests can alleviate work from testers, making it beneficial to convert manual test cases into automated GUI tests. However, automated GUI tests come with costs and drawbacks not found in manual tests. These limitations can differ between automated GUI testing tools.

Two such tools are Selenium and EyeAutomate. The tools differ in their ways of locating GUI components, with Selenium utilising underlying information about a web page and EyeAutomate relying on image recognition. For a practitioner deciding to adopt either tool, or similar ones, it is a benefit to know the strengths and weaknesses of them.

This study has investigated general differences, implementation cost, maintenance cost, return on investment, and the defect-finding capabilities of Selenium and EyeAutomate. These properties were examined by subjecting tests written in each tool to system changes using version control history. Additional capabilities were determined by using manual fault injection. Qualitative data concerning the tools and automated GUI testing were collected using interviews.

Results indicate that while EyeAutomate tests are quicker to implement than Selenium tests, they require more time to maintain. Both tools have a similar return on investment, being able to reach it within one year compared to running a manual test suite weekly. The tools are comparable when finding defects during system development, with EyeAutomate being able to find more purely graphical related defects.

Keywords:

Software Engineering, Automated GUI Testing, Element-based Testing, Visual GUI Testing, Maintenance, Return On Investment, Fault Detection

Acknowledgements

We want to thank our thesis supervisor Robert Feldt for handling our academic questions and helping us flesh out the purpose of the thesis. CANEA provided us with a place to work and easy access to coffee and fruit, for which we are grateful. We would also like to thank the people at CANEA themselves; Tomas Odin for being our company advisor when it came to questions regarding the company and the rest of the developers for participating in our two-hour interviews. Finally, we want to thank the people at Auqtus AB for their quick responses to our EyeAutomate related questions.

Patrik Haar and David Michaëlsson, Gothenburg, June 2018

Contents

List of Figures	xiii
List of Tables	xv
Glossary	xvii
1 Introduction	1
1.1 Statement of the Problem	2
1.2 Purpose of the Study	3
1.3 Research Questions	4
1.3.1 Supplementary Research Question	4
2 Background	5
2.1 Relevant Theory	5
2.1.1 Regression Testing	5
2.1.2 Automated GUI Testing	6
2.1.3 Generations of GUI Testing Tools	6
2.1.4 Element-based GUI Testing	7
2.1.5 Visual GUI Testing	7
2.2 Tools	8
2.2.1 Selenium	8
2.2.2 The EyeAutomate Family	9
2.3 Related Works	10
2.3.1 Overview of the field	11
2.3.2 Automated Testing	12
2.3.3 Element-based GUI Testing Tools	12
2.3.4 VGT Tools	13
2.3.5 Comparison of GUI Testing Tools	14
2.3.6 Defect Finding Capabilities	15
2.3.7 Implementation and Maintenance Cost	16
2.3.8 Return on Investment	17

2.4	Case	17
3	Methods	19
3.1	Motivation	19
3.1.1	Historical	21
3.1.2	Artificial	22
3.1.3	Interviews	23
3.1.4	Triangulation	23
3.1.5	Research Questions	24
3.2	Historical	25
3.2.1	Designing Tests	25
3.2.2	Setup	26
3.2.3	Execution	26
3.3	Artificial	28
3.3.1	Setup	29
3.3.2	Changes	30
3.3.3	Execution	33
3.4	Interviews	34
3.4.1	Setup	34
3.4.2	Execution	35
4	Results	37
4.1	Implementation Cost	37
4.2	Maintenance Cost	40
4.3	Return on Investment	46
4.3.1	Calculated	46
4.3.2	Infrequent Runs	47
4.3.3	Frequent Runs	49
4.4	Fault Detection Capabilities	49
4.5	Differences Between the Tools	53
4.6	Qualitative data	53
5	Discussion	57
5.1	Research Methods	57
5.1.1	Historical	57
5.1.2	Artificial	58
5.1.3	Interviews	59
5.2	Maintenance Cost	59
5.2.1	Repair Cost differences	60
5.2.2	Fluctuating Maintenance Cost	60
5.2.3	Conclusion	61

5.3	Return on Investment	61
5.3.1	Implementation Cost	61
5.3.2	Maintenance Cost	62
5.3.3	Qualitative	62
5.3.4	Conclusion	63
5.4	Fault Detection Capabilities	63
5.4.1	Test Scenario	63
5.4.2	Comparison	64
5.4.3	Conclusion	64
5.5	Selenium vs. EyeAutomate Observations	64
5.5.1	Test Implementation	65
5.5.2	PageObjects	65
5.5.3	Prior Knowledge	66
5.5.4	Locators	66
5.5.5	Data Verification	67
5.5.6	Conclusion	68
5.6	Limitations	68
5.7	Threats to Validity	69
5.7.1	Conclusion Validity	69
5.7.2	Internal validity	70
5.7.3	Construct validity	72
5.7.4	External validity	72
5.8	Contributions	72
5.9	Future work	73
6	Conclusion	75
	Bibliography	77
A	Interview Answers	I
B	Selenium example	VII
C	EyeAutomate example	IX

List of Figures

4.1	Selenium PageObjects over time	38
4.2	Maintenance sample spread	42
4.3	Total time spent per step	43
4.4	Maintenance running total	44
4.5	Histogram over repair time	44
4.6	Return on Investment (Quarterly runs)	48
4.7	Return on Investment predictions (Quarterly runs)	48
4.8	Return on Investment (Weekly runs)	49
B.1	Example of Selenium locators	VIII
C.1	EyeStudio with an EyeAutomate script	IX

List of Tables

3.1	Research methods	20
3.2	Historical - Implementation	27
3.3	Historical - Stepping	28
3.4	Artificial - Structural	30
3.5	Artificial - Graphical	31
3.6	Artificial - Layout	32
3.7	Artificial - Application	33
3.8	Artificial - Browser	33
3.9	Artificial - Execution	34
3.10	Interviews - Tool walkthroughs	36
4.1	Selenium implementation details	38
4.2	EyeAutomate implementation details	39
4.3	Implementation differences	40
4.4	Selenium stepping details	41
4.5	EyeAutomate stepping details	41
4.6	Maintenance time - Measured and extrapolated	45
4.7	Execution time - Manual and automated	46
4.8	Linear return on investment calculations	47
4.9	Unique findings - Bugs and breaks	50
4.10	Artificial result	52
A.1	Interview answers - Pretest questions	I
A.2	Interview answers - Selenium questions	II
A.3	Interview answers - EyeStudio questions	III
A.4	Interview answers - Posttest and general questions	V

Glossary

Artificial: One of the methods used in this study, which involved manually injecting faults into the system in order to determine the fault detection capabilities of the tools.

Element Locators: The identifiers used by the tests to keep track of GUI elements.

Fault detection capabilities: How good a tool is at detecting certain types of faults.

GUI Testing: The process of testing the GUI of a software.

Historical: The main method used in this study, which involved using version control history to retroactively simulate software evolution during a year.

Software evolution: The process of continuously updating software.

Test: The code that performs the instructions of a test case.

Test Case: A set of instructions that verifies that a system complies to its requirements.

Test Script: File containing ordered test instructions to execute.

Test Suite: A collection of tests.

Test Tool: The program used to create automated GUI tests. In the context of this study, test tool refers either to Selenium or EyeAutomate.

Version Control: Tracks the code changes of a system, used for collaboration and back-up purposes.

1

Introduction

Interacting with and testing the GUI (Graphical User Interface) of an application manually is often considered a mundane task. As this testing process is often repeated during the development of an application, manual testing of applications requires a lot of human resources and time. An alternative to manual GUI testing is to let a computer perform the same interactions. This is known as automated GUI testing. This way, personnel can spend time on other development activities. Automated GUI testing also enables more frequent executions of a test suite, allowing a developer to find defects earlier than through manual testing.

Despite the benefits of automated GUI testing, not all developers use it in development. There are many explanations for why this is the case. One explanation is that maintaining automated GUI tests can be difficult with tests breaking to minor changes or throwing more false positives than they should [1]. Another explanation is the often large implementation cost associated with the time-consuming work of writing the tests [2].

Since automated GUI tests are generally faster to execute, the question becomes: What is the most time efficient, manual testing or writing and maintaining automated tests? The answer to this question depends on which tool is used as there are a lot of them and they all come with different benefits and drawbacks.

There are many studies related to GUI testing and a lot of opinions about which tool to use. One of the more well-known testing tools within research is Selenium WebDriver (henceforth called Selenium) which has been compared to many other testing tools [3, 4, 5]. In contrast to Selenium, EyeAutomate with the script editor EyeStudio is a new and less established testing tool within the VGT (Visual GUI Testing) domain. EyeAutomate, being VGT-based, uses image recognition to locate elements. In contrast to EyeAutomate, Selenium relies on the structure of a web page to find elements. These characteristics makes the two tools very different when writing and running tests.

This study will compare EyeAutomate and Selenium in an industrial setting, something that the field needs more of [6]. EyeAutomate was chosen because it is a very new tool with an interesting approach, Selenium was chosen to have something established and thoroughly tested as a base of comparison. Outside of research, this study aims to provide a way for practitioners and companies to make an informed decision on which GUI testing tool to choose. Especially with regards to the maintainability of an automated GUI test suite.

This chapter of the report, Introduction, will go through the research problem, the purpose of this study and define the research questions. Chapter 2 goes through the background including explanations of relevant concepts used in this report, related research, information about the company the study will be performed at and a description of the tools under test. Chapter 3 describes the methods used to set up the study and gather data, with detailed steps describing the execution of each approach. Chapter 4 states and illustrates the results of the study. Chapter 5 discusses the results and draws conclusions based on the data, reflects on the methods used and defines any threats to validity. Finally, the study is summarised and rounded off with a conclusion in chapter 6.

1.1 Statement of the Problem

There are many areas in which GUI testing tools can differ, making it difficult to get an overview. The fragile nature of GUI testing, the opinion of the industry and the cost of implementing tests are some of the major problems related to GUI testing.

Fragility:

In contrast to unit tests, GUI tests are affected by both back-end and front-end changes. A bug in the back end can propagate to the GUI. For instance, the wrong page being displayed, incorrect output or a button that stops working. A GUI may also change frequently during software evolution. Whether it's a button moved, an image swapped or a layout changed, all of them are able to break a GUI test. These changes generates extra work for developers as they have to update the test to the new GUI. It is therefore very important that a GUI testing tool can accept some changes without breaking while still preventing any false negatives. Achieving this balance is crucial when it comes to the usefulness of a testing tool.

State of the industry:

There is a disconnect between the academic literature and the attitude of the industry. Several research studies have been performed claiming the relevance of automated GUI testing, image-based or not [7, 8, 9]. However, 58% of practitioners do not agree that automated testing improve fault detection and 80% of them cannot see how software testing could be fully automated [6]. Note that this study is based on all types of automated testing and the answers might not be the same when asked specifically about GUI testing. Nonetheless, automated GUI testing is not well established within industry and part of this is because many practitioners are not familiar to the field [6].

Cost:

Automated tests can be difficult to maintain if they are dependent on complex and volatile source code. This is especially true for GUI tests. With their fragility and frequent changes, The effort required to maintain them can exceed the effort to do the test manually. Even if the maintenance cost is tolerable, there is also a cost associated with creating the tests. This implementation cost can also be considerable. Testing tools can come with hefty licensing costs, limiting practitioners to try all that might be applicable. Perhaps even more important is the time investment required to implement a test suite of reasonable size. An investment of that size can prove a hurdle for many practitioners, forcing them to take a risk and commit to a choice. This risk would be reduced if more and better research existed within the area.

These three issues make it very hard for practitioners to include GUI testing in their test suites. The existing distrust of GUI testing within the industry makes practitioners hesitant to try it [6]. This would not be an issue if there was an easy way to try the different testing tools, but with the high implementation cost of a new test suite, this is simply not feasible. Add to this the fragility and maintenance cost of GUI tests, and the outcome is practitioners who do not dare to commit because of the high risk and uncertain results.

1.2 Purpose of the Study

The purpose of this study is to compare the two automated GUI testing tools Selenium and EyeAutomate with a focus on maintainability. The study aims for a result representative of the industry by using the version control history of a real software product to simulate the maintenance cost over a year. The tools will be examined both quantitatively and qualitatively. The study will then describe their benefits and drawbacks, analyse them and draw conclusions of how the differences affect their use. Therefore, the results of this study will be beneficial for practitioners striving to adopt automated GUI testing.

1.3 Research Questions

In order to guide this study, research questions have been formulated. This study aims to answer three research questions related to the tools and their associated costs. These research questions are:

RQ1: What are the practical differences between EyeAutomate and Selenium?

This research question will provide the basis of the comparison study as well as provide context for practitioners.

RQ2: What is the cost to maintain automated GUI tests?

The maintenance cost is the effort required to fix a test after it has broken. More specifically, the effort required to update a test to a passing state due to non-defect system changes.

RQ3: What is the return on investment for the tools?

This research question can help the practitioners decide if the tools are worth the implementation and maintenance effort. It involves gathering and comparing of implementation cost, maintenance cost and manual testing cost. The return on investment would then be the point in time where the accumulated time spent creating and maintaining an automated GUI test becomes less than the accumulated time of running the same test manually.

1.3.1 Supplementary Research Question

The goal is to answer the following question as well because it is highly relevant to the subject of the report. However, because the question can be scrutinised enough to warrant a separate paper, the focus will be more on the basics and potential outliers.

SRQ4: What are the fault detecting capabilities of the tools?

An essential property of any test is: Does it catch the bugs it should? If it is known which types of bugs or defects are the most common ones in a given project, it is important that these are covered by the chosen tool. This research question will answer which types of defects the tools can and cannot find.

2

Background

This chapter will go through relevant background and technology, prior research in the field and relevant information concerning the tools under inspection. This chapter concludes with background information about the company where this study was performed.

2.1 Relevant Theory

This section goes through and explains the most important concepts relevant to the study. These includes testing methodologies and automated GUI testing technologies.

2.1.1 Regression Testing

The source code of an application is continuously updated during software evolution. One issue with an ever-changing code base is that updates might break previously working functionality. Regression testing is the process of performing functionality tests on the source code in order to verify that the old code is working with the new changes.

Regression testing can be done in different ways. One way is manually by using human testers. Manual regression testing requires a human to perform specified test case instructions in order to verify that the application still works. For a human tester, this would take both time and resources.

2.1.2 Automated GUI Testing

Automated testing is the approach of using software to execute tests. Instead of using a human to perform instructions and evaluate responses, software is written to perform the same tasks. Automated tests on a SUT (System under test) are often created on two separate levels: source code level and GUI level. As the name implies, automated source code tests interact with a system through the source code. Unit tests are an example of source code level tests. In contrast, an automated GUI test interacts with a SUT through its GUI, the same way a user would. Automated GUI tests can be classified into two different categories based on how the tests are created: Programmable and Record & Replay.

Programmable tests: These GUI tests are defined by manually typed instructions. The tests are written as code in the supported language of the GUI testing library.

Record & Replay tests: Record & Replay (R&R), or alternatively Capture & Replay (C&R), are tests created by recording a human tester's interactions with a GUI. These interactions can then be replayed in order to mimic a user. Compared to programmable tests, R&R tests usually have a lower implementation cost but a higher maintenance cost [5].

2.1.3 Generations of GUI Testing Tools

With the GUI evolving over time, so have the tools to test it. There are currently three generations of GUI testing tools: coordinate-, element- and image-based tools, representing the 1st-, 2nd- and 3rd-generation respectively. The element-based 2nd generation has almost completely replaced the coordinated-based 1st generation due to its better stability. Hence, 2nd generation tools are very popular in industry for automated GUI testing. The image-based 3rd generation is not a new concept but has recently begun to rise in popularity, partly due to improved image-recognition capabilities.

2.1.4 Element-based GUI Testing

Element-based GUI testing, also called component-, DOM- and structure-aware GUI testing, interacts with the GUI using references to the elements of the GUI. The actual definition of an element varies between GUI implementations. For a web page, an element could be an element of the DOM (Document Object Model). For a Java Swing GUI, the element could be a reference to a Swing object.

Because it uses elements to navigate, element-based GUI testing can often perform actions in an application before it has been rendered completely. While this can cause problems in some cases, it is in general significantly faster and more precise than a human tester. However, since element-based GUI testing is bound to an application, any interactions outside of said application can be very complicated or impossible to simulate. For example, using a desktop mail client when testing a web application.

Element-based testing tools benefit from defined GUI element identifiers such as IDs, type, labels, etc. In order to narrow selectable elements, these identifiers need to be grouped and combined. Hence, in order to create effective element-based GUI tests, a programmer would need to know these identifiers and the structure of the GUI. These restrictions limit who can write element-based tests.

2.1.5 Visual GUI Testing

Using images, rather than elements, to navigate allows for complete black-box testing. No knowledge of the application is necessary apart from knowing what it looks like. Since there is no inherent limitation in the application, VGT can switch context or application without a problem, even during a test. The execution speed of a VGT (Visual GUI Testing) test is in general slower than a corresponding element-based test [3], more on par with a very experienced human tester in execution speed. However, image recognition is not perfect. In some cases it is too strict and does not find an image that would be obvious to a human. In other cases, the test is too imprecise and finds the wrong image.

2.2 Tools

There are many software tools which can automate GUI interactions. This study will evaluate two of them: Selenium and EyeAutomate. While EyeAutomate is part of a collection of different tools working together called the EyeSuite, the collection of tools provides a good basis for comparison. This section will further describe the details of Selenium and EyeAutomate, revealing distinguishing features for each tool and how they are used.

2.2.1 Selenium

Selenium¹ is an element-based tool for automating web browsers, which means that Selenium can be used to verify functionality of a web application through GUI level tests. Creating Selenium tests can be done in two different ways: through the use of Selenium IDE² or by using Selenium WebDriver³. In this study, Selenium Webdriver was used due to its lower maintenance cost [5].

Selenium IDE:

In Selenium IDE, interactions with the SUT is recorded and are stored as a Selenium script. The script can later be replayed in order to test the GUI automatically.

Selenium WebDriver:

Selenium WebDriver is an API which enables a coder to write Selenium scripts using programming language bindings. In Selenium WebDriver, identifying which element to interact with is done through locators. A locator uses the characteristics of a web element in the DOM, such as a unique attribute or a defining arrangement, to find elements. Going by the documentation as provided by Selenium HQ, the organisation behind Selenium, the locators one can use with Selenium Webdriver are:

- By ID
- By Class Name
- By Tag Name
- By Name

¹<http://www.seleniumhq.org> (Accessed 2018-06-12)

²<http://www.seleniumhq.org/projects/ide> (Accessed 2018-06-12)

³<http://www.seleniumhq.org/projects/webdriver> (Accessed 2018-06-12)

- By Link Text
- By Partial Link Text
- By CSS
- By XPath
- By Javascript

Deciding which locator to use can be complicated. The recommended approach by Selenium HQ is to use ID locators. The reasoning being that ID locators are more readable and are less demanding on performance. Precaution should be taken for auto-generated IDs as they are not necessary constant during software evolution. CSS and XPATH locators can also be troublesome as they are susceptible to DOM structure changes [3]. Some examples of Selenium locators are shown in Appendix B.

PageObject Pattern:

A design pattern which can be used when implementing tests in Selenium is the PageObject pattern. The pattern separates the GUI logic from a test to a separate class which the test can interact with. The PageObject class should contain methods which interacts with a given page using locators. A benefit with this pattern is that it reduces the maintenance time of a test suite. Though, the pattern also increases the initial implementation effort for a test suite [10].

2.2.2 The EyeAutomate Family

The EyeSuite is a collection of four programs involving VGT. These programs compose a toolset for testing applications. At the core is EyeAutomate, the program running the test scripts and enabling the automation of applications. Supporting EyeAutomate is the IDE EyeStudio and server client EyeServer, providing a custom tool for developing EyeAutomate scripts and a way to run them remotely. In another category and not based on EyeAutomate is EyeScout, an exploratory testing tool.

EyeAutomate:

EyeAutomate is the program that runs EyeAutomate test scripts. It has built-in support for image-recognition and uses customisable commands written in Java to call upon its functionality. This means that users can create their own commands if their desired use-case isn't covered by the base functionality. After EyeAutomate has run a script, it generates a report of the run with screenshots for all executed steps of the script, allowing testers to quickly locate where a problem could be.

EyeStudio:

EyeStudio⁴ is an IDE for writing EyeAutomate scripts and was used to write all the VGT scripts in this study. EyeStudio uses a "What You See Is What You Get"-like approach to display scripts, where images and commands are shown as-is to the user; what it looks like to work in EyeStudio can be seen in Appendix C. It also provides some quality of life features, including the ability to run parts of a test script and the integration with EyeAutomate result reports to highlight any crashes directly in the code.

EyeServer:

Running scripts remotely on a dedicated server can be done using the web service EyeServer⁵. Similarly as when running a script from EyeStudio, EyeServer generates a report of the result, the difference being that EyeServer also stores this report with some overhead in order to create a history.

EyeScout:

EyeScout is a tool for augmented exploratory testing. The tool can repeat prior test interactions and give suggestions of further test actions. EyeScout can be used as a regression testing tool. However, the main focus of the tool is exploratory testing. This is why this study implemented the tests using EyeAutomate instead.

2.3 Related Works

Automatic GUI testing is an area that has been increasing in popularity over the last 15 years, although the foundation was placed long before that. During the 1990's, a number of different GUI automation tools were created [11, 12]. From there it did not take long for similar tools to be used for automatic GUI testing [13]. However, the coordinate-based technology of these first tools left much to be desired in terms of stability, performance and ease of use [13]. After this, the field of GUI testing was relatively quiet until around 2005 when the emergence of the more easy-to-use element-based tools created a spike of interest from the industry. Of the more than 50 studies, in the field of GUI testing and relevant to this study, more than 80% were published after 2005.

Most of the information gathered for this literature study came from searches using the keywords VGT, 2nd generation (component-/element-/DOM-/structure-aware) GUI testing, web testing, regression testing, automated testing and the state of the industry.

⁴<http://eyeautomate.com/eyestudio.html> (Accessed 2018-06-12)

⁵<http://eyeautomate.com/eyeserver.html> (Accessed 2018-06-12)

2.3.1 Overview of the field

Regarding VGT, a large portion of the contributions to the field is written by Emil Alégroth, often assisted by Robert Feldt, using the tool Sikuli⁶ and JAutomate⁷. His contributions include the applicability of VGT in industry [14], its benefits and limitations [15] and how to successfully transition a company into using VGT [7].

The field of 2nd generation tools have been examined by many different researchers. Antawan Holmes and Marc Kellogg confirmed the usefulness of the technology in an agile workflow [16], Leotta et al. goes over the differences between scripting and recording tests [5] and Adamoli et al. compares the performance of different recording tools [17].

Filippo Ricca and Paolo Tonella have authored several studies within web testing, with topics varying from testing processes [18], testing techniques [19] and different types of locators [3].

Significant research concerning regression and automated testing related to GUI testing has been done by Atif M. Memon. As one of the first researchers to focus on GUI testing, he wrote an early evaluation of the process and potential pitfalls of GUI testing [2]. After that, he focused on regression testing and published papers about how they work for GUI testing [20], how to automate them [21] and later how to automatically repair them [22]. He has also been a part of the creation of several different GUI testing related tools such as DART, an aid for daily/nightly GUI test automation [21], TerpOffice, used for evaluating new GUI testing techniques [23], GUITAR, used for repairing GUI test suites for regression testing [22].

Something that quickly became apparent during this literature study is that there is a disparity between how academia views GUI testing and what the industry wants from it. Vahid Garousi did a systematic literature review of literature reviews within software testing and mapped what research had been made in the field [24]. He then compared the titles of different conferences to highlight the difference in approach to software testing used by the industry and academia [25]. Continuing on this track he made a survey to gather the opinion of practitioners, concluding that academia is more interested in theoretically challenging topics while industry wants the emphasis to be on effectiveness and efficiency [26].

⁶<http://www.sikuli.org/> (Accessed 2018-06-12)

⁷<http://jautomate.com/> (Accessed 2018-06-12)

2.3.2 Automated Testing

Using automated testing in practice has been explored in an experience report by Berner et al. They propose to select often run test cases for automation as these will yield the highest return on investment. It was also observed that automated tests cannot fully replace manual tests. Another finding was that automated tests are not good at finding new defects but rather defects similar to the ones the automated test cover. The fact that automated tests benefit from being run frequently in terms of maintenance was also observed in some of the cases. Berner et al. emphasises that automated testing can free time for testers, enabling more work on other tasks [27].

In a Systematic Literature Review (SLR) by Rafi et al., studies related to automated testing were mapped and practitioners' views regarding automated testing were gathered. From the SLR, some benefits with automated testing were noted such as: high test coverage, less manual effort, reduction in cost and increased fault detection. The authors also noticed some limitations with test automation such as difficulties in maintenance, false expectations and that automation cannot replace manual testing. A survey which gathered the opinions of practitioners regarding automated testing was also created. According to the surveyees, the main benefits of automation are: reusability, repeatability and effort saved. A limitation with automation would be that automation has a high initial cost which can include buying licenses or training staff. Another finding is that 80% of the surveyed practitioners don't think that software testing should be fully automated [6].

Another study of practitioners view found that context is a driving factor when selecting automation tool such as the cost or if it is open source. Many practitioners also seem to prefer more well-known tools [28].

Motivation: With the potential behind automation growing more and more, companies look for new areas to automate. This study will further examine the maintenance associated with regularly run automated GUI tests, which type of tools are preferred and which types of tests are suited for automation.

2.3.3 Element-based GUI Testing Tools

Element-based testing tools are robust to minor layout changes. Changes to the GUI code, platform or external libraries can have adverse effects [29, 30]. Li et al. states that substantial manual effort is needed for R&R testing [31].

Selenium has been explored in several studies [3, 4, 5, 16, 19]. In a Grey Literature Review by Raulamo-Jurvanen et al., Selenium was found to be the most referenced and compared tool in the assessed sources [28].

Research done by Raulamo-Jurvanen et al. has shown that Selenium is one of the more popular automated testing tools as indicated by surveys and web scraping. In the same study the prevalence of Element-based testing tools in the industry can be seen with three of the top five test execution tools being Element-based (Selenium, QTP and Rational Functional Tester) [32]. Similarly, Li et al. claims that Selenium is one of the most popular AJAX testing tools [31].

Motivation: Considering the popularity of Selenium in both research and industry, Selenium would make a good reference tool to compare with EyeAutomate. Additionally, Selenium uses information about the DOM to locate elements whereas EyeAutomate uses images. It was mainly due to these reasons which motivated the choice for Selenium.

2.3.4 VGT Tools

A noted benefit with VGT tools is that they are flexible to work with any application with a GUI [33]. This is because the tools do not need access to the code of the SUT, rather relying on captured images from the GUI alone. Due to this, visual GUI testing is robust against changes in a system's GUI code compared element-based testing [30].

Transitioning manual regression tests to automated VGT has been researched by Alégroth et al. They noted that a manual regression test that took 16 hours to run could be executed as a VGT test in one hour. Furthermore, the bottleneck of the VGT tests was the GUI of the SUT. This was due to the fact the tests often had to wait for the GUI to react in order to proceed with the test suite. Additionally, the VGT tests were able to uncover defects previously not found in the manual regressions tests [33].

The combination of 2nd and 3rd generation techniques in a tool was explored in a study by Alegroth et al. [34]. In this study it was determined that 3rd generation technique reports fewer false positives compared to 2nd during acceptance testing. Although for system testing, the opposite was observed in that 3rd generation technique reports more false positives than 2nd generation. The authors of the study proposed that a combination of the two GUI testing techniques could mitigate the studied drawbacks.

The process of automatically generating Visual GUI tests from element-based test suites has been researched by Leotta et al. This resulted in the tool PESTO which can transfer test cases written in the 2nd generation tool Selenium WebDriver to 3rd generation test cases using Sikuli [35].

A case study at Spotify by Alégroth and Feldt investigated how VGT fares in industry long-term. Some of the benefits observed from using VGT in practice is the technique's robustness, defect finding ability during regression and the ability to share test script logic between different versions of the SUT. There were also drawbacks associated with the technique. An obstacle to the technique is that VGT does not support non-deterministic test data. In this case, the SUT relied a lot on dynamically rendered content, which the VGT tool could not verify. Another drawback is the maintenance cost of recapturing images. This was particularly noted when the GUI graphics were removed or changed [7].

Motivation: Alégroth et al. acknowledges the need for more studies comparing VGT with other GUI-based techniques in an industrial context [15]. Many of the studies in VGT research has been centred around Sikuli [3, 7, 14, 15, 33, 34, 36, 37]. Further studies of other VGT tools could reveal new benefits and limitations of using VGT. To the authors' knowledge, there are currently no studies concerning EyeAutomate.

2.3.5 Comparison of GUI Testing Tools

A comparison study of two VGT tools has been done by Börjesson and Feldt. In this study, Sikuli and an undisclosed VGT tool were evaluated when used on a system developed at SAAB. Between the tools, there were no statistically significant differences in regards to development time, execution time and lines of code (LOC) of the tests. It was also determined that Visual GUI tests can overcome some R&R test limitations such as needing access to the code of a system or the tests being strongly tied to a GUI component. The authors also advocate that more studies are needed concerning the maintenance cost of a Visual GUI Test suite as a system evolves [36].

Leotta et al. have compared DOM-based locators with visual locators in order to compare the required number of locators, robustness, implementation, maintainability and execution time between the two approaches. The result of the study concludes that DOM-based locators are in general more robust compared to visual locators. Another finding of the study was that developing and evolving tests using DOM-based locators were less costly than using visual ones. The VGT tests took a longer time to execute, though the difference wasn't dramatic [3].

The VGT tools JAutomate and Sikuli were evaluated at the company HAVELSAN by Garousi et al. Some limitations of the tools were indicated such as difficulties with using too small image locators and running the same VGT script on computers with different display resolution. While there were differences regarding features between JAutomate and Sikuli, such as the R&R feature found in JAutomate, there were no statistically significant differences in regards of robustness and repeatability. Concerning test development effort, the authors noted that the test-code reuse pattern reduced development effort as it enabled a coder to reuse existing test code. Based on the results, it was determined that JAutomate slightly suited the needs of HAVELSAN better than Sikuli [37].

Motivation: To the researchers' knowledge, most of the comparative studies of GUI test tools have been limited to tools within the same generation of GUI testing technology. An exception is the study made by Leotta et al., which compares the 3rd generation automation tool Sikuli with the 2nd generation tool Selenium [3]. This study aims to further this body of knowledge by instead comparing Selenium with the newer 3rd generation test tool EyeAutomate.

2.3.6 Defect Finding Capabilities

Because GUI tests are prone to false positives [13, 30, 37], the accuracy, or fault-detection, appears to be assumed as high and are not given much attention in research. However, there are still some papers taking it into consideration. The study made by Memon and Xie claimed a high accuracy with their 2nd generation tool DART [38], with false negatives being drawbacks with the tool rather than inherent limitations of the technique. In another paper Alegroth et al. presents how VGT can be used to find bugs which are difficult to replicate through manual means [39]. Alégroth et al. claims that VGT is actually more capable of detecting faults than manual testing due to its low execution cost and speed [15], allowing the frequently run tests to catch faults that only occurs occasionally.

Motivation: Some fault detection research has been done with GUI testing but it has been limited to older tools or based on specific aspects. An aspect that has not been covered is what types of faults a 3rd generation tool such as EyeAutomate actually can handle. Garousi et al. has a similar idea and has stated it as future work to investigate the difference in fault detection effectiveness between different VGT tools [37].

2.3.7 Implementation and Maintenance Cost

One of the first costs encountered when considering automated testing is the implementation cost, because, in contrast to manual tests, automated tests have to be implemented before they can be used. This initial implementation cost can be high [6], perhaps too high to replace manual testing if the tests are run infrequently [29]. With VGT tools the primary cost with writing the test scripts is related to the effort of making the scripts robust to unexpected system behaviour [36], otherwise the tests would be too fragile and generate false positives.

The robustness of the tests is key since the implementation cost is not as important as the maintenance cost, as stated by Berner et al. [27]. Although the paper by Berner et al. referred to automated testing in general, the same conclusions seem valid for automated GUI testing as well. Leotta et al. used the 2nd generation GUI testing tool Selenium and also concluded that maintenance has a bigger impact than implementation because of the large number of times the tests are run [5]. In the same paper, the authors also noted that different techniques of creating the tests, Record & Replay and programmable, affected both implementation and maintenance cost. R&R had a lower implementation cost and a higher maintenance cost, causing the authors to favour the more technically challenging approach of manually writing the test scripts.

With one of the biggest obstacles to automated GUI testing being the maintenance of the test scripts, attempts have been made to automate even this, with varying successes. Atif M. Memon created a tool, GUITAR, and automatically repaired many broken 2nd generation scripts successfully [22]. When he later teamed up with Alegroth et al. to apply GUITAR in a VGT context it proved difficult and not applicable in practice [34]. Coppola et al. have researched the causes of test maintenance in Android projects. In their study, 27 causes for why an automated GUI test needs maintenance were classified into different categories. Some identified categories were: Test code changes, Application code change, GUI interaction change, GUI views arrangement, View Identification, and more. While the study was performed on Android projects, the authors believe that the defined test maintenance causes may be used for other GUI-based software types [40].

Motivation: While many studies have looked at maintenance cost of GUI testing, the analysis was always based on different versions of the software. Usually, these data-points were few and far apart. Different from these approaches, this study will look at the maintenance cost over time with weekly samples during a year. The SUT will be an established and active product, which is more in-line and usable by the industry.

2.3.8 Return on Investment

Alégroth et al. have researched the return on investment for VGT test suites using data gathered at Siemens and SAAB. They found that a positive return on investment is possible for the introduction of a VGT suite. However, maintenance cost can still be substantial when compared to the time spent on V&V (Verification and Validation). The time needed to come to a positive return on investment is consequently dependent on the amount of V&V done before the introduction of VGT [8].

The viability of 2nd generation C&R tools has been examined in research. The effort cost for such a tool was higher when compared to the effort of performing manual regression testing [29]. In a comparison of the implementation and maintenance costs between Selenium IDE and Selenium WebDriver tests, it was determined that two major releases are needed before a Selenium WebDriver test suite becomes more convenient than a corresponding C&R one [5].

Berner et al. have noted that many organisations have the wrong expectations of automated testing, with many of them hoping for a very short return on investment since adopting automated testing [27]. Berner et al. also urges that the freeing of human resources and shorter release cycles should be taken into account when opting for automated testing.

Motivation: To the researcher's knowledge, there have been no studies comparing the return on investment for both a 2nd- and 3rd generation test suite against the effort of performing the test cases manually.

2.4 Case

This case study will be done in collaboration with the company CANEA at its Gothenburg office. To get a good coverage the topics were based on those suggested by Kai Petersen and Claes Wohlin [41], although in a simplified version.

The product is CANEA ONE, a web-based business management tool. The first framework, known then as CANEA Framework, was launched in 2002 as a local desktop application. Throughout the years, it has been developed into what is now known as CANEA ONE. In 2007, it was launched as a web-based application. In 2012, CANEA ONE was made globally available.

CANEA ONE is a large product composed of many different languages, the majority being written in C#, Type-/JavaScript, and HTML. The C# code alone is more than 250 000 lines of code and the web-application has more than 100 unique pages. The source code is automatically tested through roughly 2700 unit tests. The UI testing has only the most basic automatic testing through JMeter and CodedUI. Most of the UI testing is done manually by a team of testers.

The development process at CANEA is an agile work method based on SCRUM and Kanban. Each sprint is set to last three weeks, with major releases to customers every third month. The code is automatically built and tested twice a day on a TeamCity server during the sprints, with manual testing being done continuously on the active build. Additional and more in-depth testing is performed before each release.

The Organisation CANEA has offices in Malmö, Stockholm with the main office in Gothenburg and consists of three branches: Consulting, Training and IT solutions. The branch IT solutions develops and maintains the product CANEA ONE, which will be the subject of this study.

The People in the product developing part branch of the company consists of testers and the usual personnel connected to SCRUM: Product owner, SCRUM master, and developers. In Gothenburg, the team consists of one product owner, one tester and nine developers. The developers are relatively young with 0 to 10 years of experience.

The Market CANEA operates on is that of business management software suites. CANEA ONE is used by more than 200 organisation. Among these are *Sandvik*, *Göteborg Stad* and *Husqvarna*. In the same market, there are other similar, competing products such as Microsoft Project, Podio, and Basecamp. CANEA ONE is a highly configurable, on-premises software. As such, setup and configuration are significant procedures in the CANEA ONE usage process. A customer using CANEA ONE can request support for these steps from CANEA.

3

Methods

The testing tools EyeAutomate and Selenium will be tested in a case study with three different research methods: historical testing, artificial testing and interviews. The historical testing will be based on old versions of CANEA ONE and will be focused on measuring the maintenance cost. Determining the defect finding capabilities of the tools is the purpose of the artificial testing. Qualitative data about the tools will be captured by interviews. These will then be compared with the cost in work-hours of manual testing to answer the research questions. An overview of the study can be seen in Table 3.1.

3.1 Motivation

The main focus of this study is maintenance for test suites, which is dependant on changes and repairs over time. Tests would need to be written and then repaired as changes are introduced into the system. As with any time-related data gathering, a long time period and frequent samples are the best. In this case, a longitudinal study over a year or more with daily samples would be the optimal solution. This is not feasible due to the time constraint of four months for the entire study.

The use of the company's version control history and focus on the industry could merit the use of a case study or action research. However, because the version control history is based on what has already happened and does not rely on humans, a more controlled method such as an experiment is also a possibility. Using action research wouldn't suffice since one of the goals of the study is to get a realistic estimation of maintenance costs over time. An action research study strives to improve the process during the study, thereby skewing the end result. A case study could work, but with the small number of variables, it makes more sense to use something more controlled.

Table 3.1: Summary of the the research methods of the study.

Historical - Quasi-experiment	
Purpose	Estimating implementation cost and maintenance cost over time.
Execution	<ol style="list-style-type: none"> 1. Go back one year in version control. 2. Implement a number of carefully selected test cases. 3a. Step a week forward in version control. 3b. Run tests in both tools. 3c. Repair any broken tests. 4. Repeat 3 until 1 year has been examined.
Notes	If any bugs are found they will be recorded and circumvented (if possible) in order to keep running the tests, even if the bug remains.
Artificial - Experiment	
Purpose	Examining the accuracy/fault detection of the tools, covering common cases in a strict manner.
Execution	<ol style="list-style-type: none"> 1. Define around 20 different types of changes (visual, bugs or minor). 2. Select one of the historical tests deemed to cover a large area. 3. Introduce the changes from 1 into the code one at a time and run the test in both tools. 4. Record which changes broke/were caught by which tool.
Notes	While similar data can be gathered from the historical, the artificial allows for wider coverage of changes and a more direct comparison of the tools.
Qualitative - Semi-structured Interviews	
Purpose	Forming a qualitative opinion on the tools and the validity of Automated GUI Testing.
Execution	<ol style="list-style-type: none"> 1. Teach the interviewee the tools. 2. Have them create a test with the tools. 3. Hold an interview about their experience and opinions.
Notes	Most interviewees will be developers due to the technical requirement of Selenium, but not all.

An experiment would not work due to the inability to directly control the independent variable, which is undesirable to do in this case anyway. Instead, a quasi-experiment would be ideal. If done as an experiment, the independent variables would be the tests written by the researchers and the changes to the code introduced into the system. The writing of the tests is not a problem but the new code would then have to be introduced randomly. Since it is desirable for

the new code to emulate normal development, the code needs to stay unmodified and changes need to be in iterative order. This limits the usage of randomisation. Therefore, the best alternative would be a historical quasi-experiment where the introduction of new code would be based on version control, not randomly.

In order to properly highlight and triangulate the differences between the tools, an examination of their capabilities and usability would be highly beneficial. To get a complete image of the tool's capabilities, a list of their features would have to be created and every shared feature examined in detail. This type of in-depth analysis is outside the scope of this study. However, by focusing on an essential function of any testing tool, finding defects, the scope is reduced to a manageable size. While the maintenance focused quasi-experiment can answer how often the tests break, it does not provide a clear answer to what types of changes the different tools can handle. Therefore, a quantitative second examination using artificial changes should be performed in order to list which of these changes break the tools. Since this artificial examination could be highly controlled, a small experiment would be the best choice.

One area regarding the comparison of the tools which will not be fully covered by the historical and artificial examinations, is their usability. Since the usability is a subjective measurement, it is better analysed using qualitative measures rather than quantitative ones. To cover this and other qualitative aspects of the tools, it was deemed that the opinions of the researchers were not enough. To reduce bias, interviews would be performed with the developers at CANEA.

3.1.1 Historical

The goal of the historical testing is to estimate the maintenance cost of automated GUI testing over time, including any differences between the tools. To estimate the cost over time, changes need to be introduced over time as well. The most obvious way of doing this is to let the developers work as normal while the researchers write and analyse the test results in parallel. However, it would only be possible to have a short period under examination due to the time restrictions of the study. It would also be a potential threat to validity in that the developers would be aware of the study and could change their behaviour. By instead using version control history it allows for a longer period to be studied in the same time frame, since the data gathering is not limited to the normal development pace.

Basing the study strictly on release versions of the product would not suffice. Although the versions are in chronological order, they do not provide the consistent intervals required to estimate the cost over time. Neither will they fulfil their purpose of finding bugs early because the majority of bug will most likely be found, and fixed, between releases. Therefore, the jumps through the version control history should be done in time increments, on the active development branch, instead of jumping between release versions. Another benefit of this method is that it simulates the way the tests would be used, at regular intervals as regression tests.

3.1.2 Artificial

The goal of the artificial testing is to get objective, quantitative numbers about the fault detecting capabilities of the tools. When it comes to gathering objective measurements, using an experiment-like approach is usually the first thing that comes to mind. This is with good reason because an experiment has established methods for setting up and defining these types of comparisons. The variable requirements of an experiment, only changing one or a few variables while keeping the rest constant, also fit the purpose of the artificial step. Therefore, it was deemed that a small experiment was the best fit as a research method for the artificial step.

In order to test if a tool detected a fault, the test would need to be constant while a fault was introduced in the code. The question then becomes which types of faults to use and how these faults are to be introduced into the code.

To decide which types of faults to introduce, these faults had to be defined. First, it was decided that not only faults would be introduced but also normal changes that occur during development. It was then discussed if the types of changes would be decided by the researchers, by some sort of classification from research or by taking the most common type of changes made at CANEA. In the end it was decided that the common changes at CANEA would be used and complemented by classifications from research [40], mainly to get a relevant but still wide coverage of changes.

The changes could be introduced either manually or by a tool through mutation testing. While mutation testing could give the correct type of results, it is limited in the complexity of changes it can do. In order to get a more exhaustive list of faults the tools could handle, it was decided that a curated list of changes were to be introduced manually.

3.1.3 Interviews

The goal of the interviews is to get a qualitative impression of the functionality, maintainability and overall viability of the tools. To do this it was decided to gather the opinions of the developers at CANEA. This could be done through a survey, structured-, unstructured- or semi-structured interviews.

There is a limited time each developer could reasonably spend on learning the tools, but they have to understand them. A survey would not be the best choice; since the developers have little to no experience with the tools, they would need to be taught the tools before they could give their opinions on them. Because this time is limited there is a large chance of misunderstandings. If the developers are then given a survey without any possibilities of clarification, they could base their answers on false assumptions.

A semi-structured interview fit the situation the best. In the interview, the developers would be able to ask questions to the researchers during the data gathering, clearing up any misunderstandings. Because there should be a basis for comparison between the answers of the developers, there should be some structure to the interviews. On the other hand, follow-up questions can catch the more personal opinions that often arise while trying a new tool. The best fit for this case would then be a mix between a structured- and an unstructured interview, which is the definition of a semi-structured interview.

3.1.4 Triangulation

Triangulation has been a driving factor when deciding which research methods to use. Relying on one research method to answer the research questions would be risky as any flaws in the research method would affect the results. Gathering data from multiple methods and sources will mitigate the potential risks found in a research method. Observing the same phenomena from the results of the other methods allows for more confident conclusions. The research methods will gather both qualitative and quantitative data. With the different types of data, a more diverse answer to the research questions can be given.

3.1.5 Research Questions

The methods in this section will answer the research questions as seen in section 1.3.

RQ1: What are the practical differences between the EyeAutomate and Selenium?

RQ2: What is the cost to maintain the tests?

RQ3: What is the return on investment for the tools?

SRQ4: What are the fault detecting capabilities of the tools?

The historical method will gather data to answer: RQ1, RQ2, RQ3 and in some capacity SRQ4.

Both during the implementation of the tests and the stepping through version control history, any differences between the tools will be noted for RQ1. While stepping, it will be timed for how long it takes to handle and fix each test, giving all the data necessary for RQ2. This data will then be used to calculate the return on investment for RQ3. Finally, which types of bugs were found and what caused the tests to break can be used for SRQ4.

The artificial method will provide answers to RQ1 and SRQ4.

The data gathered from the test runs will help answer supplementary research question SRQ4. Using fault-injection, the fault-detecting capabilities of the tools can be determined. Since both tools are examined, any differences found can also be used for RQ1.

Interviews will answer research questions RQ1 and RQ3.

As part of the interviews, each interviewed person will create GUI tests in each of the tools. The interviewees can then provide an industrial perspective on the tools after trying them, answering RQ1 and giving their opinions on RQ3. The interviews will provide qualitative data as opposed to the quantitative data of the other methods.

3.2 Historical

Historical testing involves going back to an earlier state of the code using version control, and implementing GUI tests at that point in time. From that point, the system will be updated using later commits in order to recreate the actual changes to the system. This allows data gathering from a time-period longer than the study duration of four months. The benefit of this approach is that the changes would be based on industrial data. A maintenance cost can be estimated by measuring the cost for correcting the tests that break between commits.

Normally, test-results are only analysed if they fail, often due to the cost of manual labour. However, this approach would miss any false negatives. A way of finding potential false negatives is through cross-comparing the run results. With a comparison between two tools being one of the main objectives of this paper, any deviations between the tools will be investigated.

3.2.1 Designing Tests

In place at CANEA are rigorous testing protocols for the current manual GUI testing. These will be used as the basis for comparison between manual testing cost and automatic testing cost. Some of the defined test cases in the manual suite will be implemented as automated tests using each tool. These tests would provide an estimation of the cost to make a transition to automated tests.

Automated GUI tests can be created in many different ways. Two tests that are syntactically different can achieve the same end result. Though, during the evolution of a system, the tests could have different results. To ensure robustness of the tests, guidelines will be followed in order to avoid common pitfalls.

Selenium:

The PageObject pattern will be used when designing test cases. This choice was motivated in that the PageObject pattern separates test logic from page logic and, therefore, reduces the effort to repair broken tests [10].

Language and Framework:

The Selenium tests will be written in C# using Selenium WebDriver bindings for C#. The choice of language was decided due to familiarity with the language at CANEA. The choice of framework subsequently fell upon .NET Framework 4.7.1 for similar reasons.

Selenium WebDriver does not include test assertion functionality on its own. For most Selenium tests, a unit test framework is needed. For .NET framework, there are mainly three popular unit test frameworks: xUnit, NUnit and MSTest. These test frameworks have similar capabilities and functionalities. For the Selenium tests it was decided that NUnit would be used. This decision was motivated in that NUnit is used at CANEA for unit testing.

EyeStudio: The guidelines found at the EyeAutomate webpage¹ will be used when designing tests scripts in EyeStudio. Development guidelines presented by Alégroth and Feldt will also be taken into consideration [42]. Some suggested actions when constructing tests are to use delays when needed, handle failed commands, divide the test script into steps using *begin* blocks and using different recognition modes depending on the locator image. Test functionality can be extracted to separate test scripts in order to enable reuse, also called modularisation. In some cases, there can be minor differences between test steps except for some input variables, for instance, a test logging in different users. Consequently, whenever necessary and possible, modularisation will be used to keep the tests smaller and more maintainable.

3.2.2 Setup

Both the EyeSuite and Selenium tests need to be run frequently and without human supervision. This requires an easy way to run the test suites consistently and with good, verifiable reports. These requirements were met for EyeAutomate through its HTML report functionality. For Selenium, the tests were written as unit-tests and reports were generated using NUnit through Visual Studio.

3.2.3 Execution

The historical method has two main phases: the implementation phase and the stepping phase. The first step of the implementation phase is to select which tests to implement based on a set of well specified manual test cases. The second step is then to decide who of the researchers to implement which tests. Finally, the tests will be implemented and data will be gathered. This process is described in more detail in Table 3.2.

¹<http://eyeautomate.com/documentation.html> (2018)

Table 3.2: The implementation phase of the historical testing. Where applicable it is shown which type of data is gathered in which step.

Historical - Implementation		
Choosing test cases		
1.	CANEA choose a few test cases with good coverage.	
2.	Researchers select a few of the remaining test cases randomly.	
3.	Go back one year in the version control history of CANEA ONE.	
4.	Perform the tests manually according to their specifications.	Data gathered
		Time to perform the tests
Dividing the test cases		
5.	The test cases chosen in step 1 are implemented by both researchers in both tools together.	
6.	The test cases chosen in step 2 are implemented by one researcher in one tool, then by the other researcher in the second tool. Who uses which tool is balanced between the researchers.	
Writing the tests		
7.	All the tests from step 1 & 2 are implemented according to 5 & 6.	Data gathered
		Implementation time
		Lines of Code
		File size
		Personal impressions

The second phase, the stepping phase, is the main part of the historical test. It covers a year of development with weekly data points, with the exception of the first and last week where samples will be taken every day. The tests are run and analysed every iteration with any broken tests being repaired before continuing. The stepping flow can be seen in more detail in Table 3.3.

Table 3.3: The stepping phase of the historical testing. Where applicable it is shown which type of data is gathered in which step.

Historical - Stepping		
Stepping between versions		
1.	Go forwards one week (one day for sample weeks) in version control.	
2.	Make sure the system builds correctly.	
3.	Run the test suites. Handle tests according to their status: nothing for passed tests, step 5 for failed ones.	Data gathered
		Test pass or fail
		Personal impressions
4.	Repeat from step 1 until done.	
Handling failed tests		
5.	Determine what caused the test to fail and categorise it. Do 6, 7 or 8 depending on the type.	Data gathered
		Analysis time
		Type of failure
6.	Bug was found: Record the bug and, if possible, create a workaround to run the rest of the test.	
7.	Occasional crash: Fails caused by unknown or unrelated events are re-run.	
8.	Breaks: The system has changed and the test needs to be updated.	Data gathered
		Time to fix

3.3 Artificial

The artificial research procedure is an experiment for measuring the capabilities of each tool. More specifically, which kinds of faults the tools can handle. Observing how the tests handle introduced defects reveals some of these capabilities. Whether the tests pass or not allow for robustness to be tested and decide which areas of the GUI a given tool is best suited for. Therefore, using the data gathered from this procedure, conclusions regarding the fault-detecting capabilities can be drawn.

The artificial testing is also a complement to the historical procedure. Changes which did not emerge during the historical stepping phase can be injected by hand into the system, in order to evaluate the capabilities of the tools.

3.3.1 Setup

A suitable test from the historical test suite will be chosen to test the introduced changes. The type of changes to be applied will be based on which changes are the most common at the company, complemented with changes used in other research.

As the artificial phase can be considered an experiment, the formal hypothesis and variables can be defined as follows:

Hypothesis: *There is a difference in the type of changes EyeAutomate and Selenium can catch.*

Splitting the caught changes into defects and harmless updates can determine the capabilities of the tools. If a testing tool is too narrow and does not catch any defects, it is useless. On the other hand, a tool that is too sensitive will catch many of the harmless changes as well, leading to many false positives.

Independent variables: *Sections of the code in a specific part of the system and web browser.*

The sections of code will be changed in several different ways, some harmless, others clearly breaking the functionality of the system. Before beginning a new change, the last one is removed and the system restored to its original state. Therefore, there will only be one change in the code at any given time. For the different browsers the system will not be changed, but rather the environment the tests are run in.

Controlled variables: *The tests, hardware and the parts of the system which are not subjected to the changes.*

Most notable of the controlled variables is the test. It will be the same test used and refined during the historical testing, and will not be changed at all during the entirety of the artificial testing. Moreover, the experiment will be performed on the same computers so the hardware will not change. Neither will the majority of the system because the changes introduced will be small and limited to a specific part of the system.

Dependent variable: *Whether the test result matches the expected behaviour.*

Although this is the only data needed to answer the hypothesis, any other findings will be recorded in order to relate it to the rest of the study.

3.3.2 Changes

Tests can fail for different reasons. As seen in Related Works, many studies have explored this subject before. The categories listed in this section are derived from previous studies, experiences from CANEA and the authors' experiences from working with the tools. All of the proposed changes have an expected test outcome.

Where in the system a change will be injected is determined by the locators used by the tests. A list of locators shared between the test tools will be created. From this list, which locator or series of locators to change in the system will be selected at random.

Structural changes: These changes consist of changes to the DOM. Some of them will change the visual appearance of the GUI while others won't. Selenium should be susceptible to these changes as it locates GUI elements using the DOM as seen in prior studies [3, 16]. The structural changes used for this study can be seen in Table 3.4.

Table 3.4: Structural changes

Structural Change	Desired Test Outcome	Notes
Change the tag type of element targeted by a test while keeping visual appearance and functionality.	Pass	The innermost DOM-element in the locator will be changed to a different type.
Surround element targeted by a test with <i>div</i> tags.	Pass	The innermost DOM-element in the locator will be surrounded with tags.
Move element targeted by a test up the DOM hierarchy.	Pass if functionality is kept, otherwise fail.	The innermost DOM-element in the locator will be moved to up to its parent's level.
Remove element targeted by a test from the DOM.	Failure	The outermost DOM-element in the locator and its children will be removed.

Graphical changes: A graphical component in a GUI may change in appearance during software evolution. This category can be seen in Table 3.5 and will capture these changes to the GUI. VGT tools such as EyeAutomate are fragile to these changes. In contrast, Selenium, which is an element-based tool, should be indifferent to graphical changes [42].

Table 3.5: Graphical changes

Graphical Change	Desired Test Outcome	Notes
Major change to the size of a GUI component targeted by a test.	Failure	Done by using CSS.
Minor change to the size of a GUI component targeted by a test.	Pass	Done by using CSS.
Change the text resource of a GUI component targeted by a test.	Failure	Sets the text path resource to another randomly selected text resource in the system.
Change the image resource of a GUI component targeted by a test.	Failure	Sets the image path resource to another randomly selected image in the system.
Hide a GUI component targeted by a test	Failure	Done by using CSS.

Layout changes: The suggested changes in this section are changes which modify the components of the GUI. This includes adding a new component to the view or removing an existing one, which are changes that can occur during software evolution. For some changes, it would be desirable that the tests should still pass, such as the addition of a distinct view component. For other changes, it would be desirable that the test fails, like when a view component covers a locator.

This category takes inspiration from the 'GUI views arrangement' classification [40]. Selenium and other element-based automated GUI test tools should be robust against these changes [29]. The layout changes to be used in the study are seen in Table 3.6.

Table 3.6: Layout changes

Layout Change	Desired Test Outcome	Notes
Add GUI component not targeted by test.	Pass	The component will be visible and close to the component used by the test.
Remove GUI component not targeted by test.	Pass	The removed component will be close to the component used the test
Hide a GUI component targeted by test behind another component.	Failure	Done by using CSS.
Change location of a GUI component targeted by test significantly.	Failure	Done by using CSS.

Application changes: These are changes to the back end of the system, concerning the logic of the system. Even though a GUI test is created in order to verify the GUI, it would be desirable if a GUI test could detect application level logic defects as well. These type of changes have also been used by Coppola et al.[40].

Where the change will be injected into the system is tied to the locators of the tests, if applicable. Given a locator and the corresponding interaction with the system, the change will be injected in the code responsible for that interaction. The full list of application changes can be seen in Table 3.7. For the *Several shorter suspended responses* change, a series of randomly selected, applicable locators will be chosen.

Table 3.7: Application changes

Application Change	Desired Test Outcome	Notes
Injected internal error.	Failure	Generated by adding a severe code error in the SUT.
One long suspended response.	Failure	A sleep statement waiting longer than the timeout cutoff of the tests.
Several shorter suspended responses.	Pass	Several sleep statements waiting shorter than the timeout cutoff of the tests.

Browser changes: Running the test in a different web browser can be achieved in both Selenium and EyeAutomate. EyeStudio even supports some functionality of Selenium by default, including the ability to launch specific browser drivers. However, the GUI test case has been implemented to run on a specific browser, the web browser Chrome. As such, browser-specific defects could emerge when the GUI is run on a web browser other than Chrome, which would then be caught by this test. The browsers to test are listed in Table 3.8. In contrast to the other changes, the browser changes are not tied to some specific locator.

Table 3.8: Browser changes

Browser Change	Desired Test Outcome	Notes
Edge	Pass	Using Microsoft Edge 41.
Firefox	Pass	Using GeckoDriver 0.19.1 & Firefox 59.0.2.
Internet Explorer	Pass	Using Internet Explorer 11.

3.3.3 Execution

The procedure for the artificial experiment is based on running a test written in both tools while introducing changes into the system. More specifically, for each proposed change as defined in subsection 3.3.2, the instructions in Table 3.9 will be performed. With 19 changes, the procedure will be repeated 19 times. The cause for why a test does not pass or fail due to an injected change will be collected as well.

Table 3.9: Artificial - Procedure for each change. Where applicable it is shown which type of data is gathered in which step.

Artificial - Execution	
1.	If applicable, randomly select a shared locator between the tests.
2.	Modify the system or environment to introduce the change.
3.	Run the test in Selenium.
	Data gathered
	If the test failed or not.
4.	Remove any added data from the system.
5.	Run the test in EyeStudio.
	Data gathered
	If the test failed or not.
7.	Reset the system or environment to its original state.

3.4 Interviews

To get a qualitative impression of the tools beyond the opinions of the researchers, interviews will be held with personnel at CANEA. The interviewees will consist of personnel working at the software development department at CANEA, both software developers and testers. The interviewees will be taught the tools individually and then get to implement a short test in the system. After the interviewees have experienced both tools, the interview is held. During this interview, the questions will focus on the tools used but also on their view on automated GUI testing in general.

3.4.1 Setup

To make sure the test is of a reasonable size and the questions are manageable, a pilot interview will be held. This pilot will do the full test as normal, but will also get questions on the interview itself. Depending on the pilot interview, adjustments will be made to improve the following interviews. Although the answers on the regular questions for the pilot will be recorded, they will not be used for the study; the study will be based on the interviews performed after the pilot has vetted the procedure.

The coding during the interview phase will be done on the researchers' computers. The test environment will be prepared before every interview to have the tools prepared, test instructions open and the SUT running.

3.4.2 Execution

The execution of each interview will be split into three parts: tool walkthrough, test writing and questions. The first two parts will be repeated for each tool resulting in a total of five steps. The order in which the tools are taught alternates in order to mitigate bias in answers since the test case stays the same for both tools. To make sure that every interviewee gets the same walkthrough, the process used was defined into the steps shown in Table 3.10.

After the interviewee has had an explanation of the tool, the testing begins. The task will be the same for every interviewee and between each tool. Though the task will be short, it will also be wide enough that it requires both usage of and additions to the existing test suite. The purpose of the test is for the interviewee to form an opinion of the tools by using them. Because the opinion might be affected by which tool the interviewee starts with, the starting order will be balanced when looking at the whole group. Another factor that might affect the interviewee's opinion of the tools is how fresh in memory the tool is. This memory depends both on how long ago they used the tool and on how much time they spent with it. To keep a reasonable length on the interview, the time spent on each test will be limited to 30 minutes; if an interviewee is getting close to, or going over, the time limit, they will get assistance from a researcher.

When the interviewee has tried both tools and formed an opinion, the interview will be performed. It will be a semi-structured interview focusing on the interviewee's perception of the tools and how well they could be integrated into the company. The interview aims to be objective; the researchers avoid leading questions and only help with ambiguity or misunderstandings of the questions.

3. Methods

Table 3.10: This table shows what was gone through during the walkthroughs, the order is of lesser importance.

Tool walkthroughs	
EyeStudio	Selenium
Show the IDE	
Script area	Visual Studio
Selecting commands	nUnit Tests
Structure	
Tests	Tests
Images	PageObjects
Writing Tests	
Images	C#
- Capturing images	Selenium library
- Re-capturing	Test structure (setup/teardown)
- Set focus point/area	Calling/Using PageObjects
Begin, End and Catch	Assertions
Calls to other scripts	Locators
Browser operations	Finding elements
Check-waits / Sleep	Waits / Sleep
Running tests	
Reports	Stack trace
Debugging	Debugging

4

Results

This chapter contains the results gathered from the research methods described in chapter 3. The first five sections look at the quantitative data gathered during the study. It goes through the implementation- and maintenance cost which is then used to calculate a return on investment. Following the return on investment is the fault detection capabilities of the tools and a summary of their major differences. The last section is about the qualitative data collected from interviews.

4.1 Implementation Cost

All implemented tests were chosen from a set of critical test cases called GXP tests at CANEA. The GXP tests are carefully specified test scenarios normally used for manual testing, numbering 20 in total. The test cases contains written instructions and expected behaviour, making them ideal to transfer to automated GUI tests. Test cases one, two and three were all chosen by employees at CANEA as cases which covered important but varied areas of the system. These were implemented together by the researchers. The remaining test cases were picked, as time allowed, at random from the remaining test cases. These random tests were implemented individually by the researchers in an alternating fashion; one researcher implemented two EyeAutomate tests and one Selenium test, two Selenium test and one EyeAutomate test for the other.

The implementation effort of Selenium was split up into two categories, tests and PageObjects. The results can be seen in Table 4.1. Of particular note is the ratio between the test LoC and the PageObject LoC. This difference is further examined in Figure 4.1 which shows a trend of a higher percentage of effort being placed on the parts unique to the test instead of the common PageObjects. This is most likely due to the reusing of PageObjects as the test suite grows.

4. Results

Table 4.1: The implementation time, size in lines of code (LoC) and size on disk for the Selenium tests. Lines of code are divided into test specific code and common PageObject (PO) code. Both the full LoC of the files and the strictly trimmed LoC from Visual Studio 2017 were given. LoC and file size are not used in this study, but was included as a base of comparison between other studies in the field. The tests are in order of implementation.

Selenium	Impl. time (s)	LoC tests	LoC tests (trimmed)	LoC PO	LoC - PO (trimmed)	File size Additions (KB)
Test 1	41748	113	55	631	145	24.2
Test 2	3201	50	12	47	9	2.8
Test 3	25162	169	119	543	139	26.7
Test 4	23900	194	107	492	92	25.1
Test 5	30763	221	115	332	71	23.2
Test 6	12320	107	49	100	30	8.1
Total	137094	854	457	2145	486	110.4
Average	22849	142.3	76.2	358	81	18.4

Selenium - Tests vs PageObjects

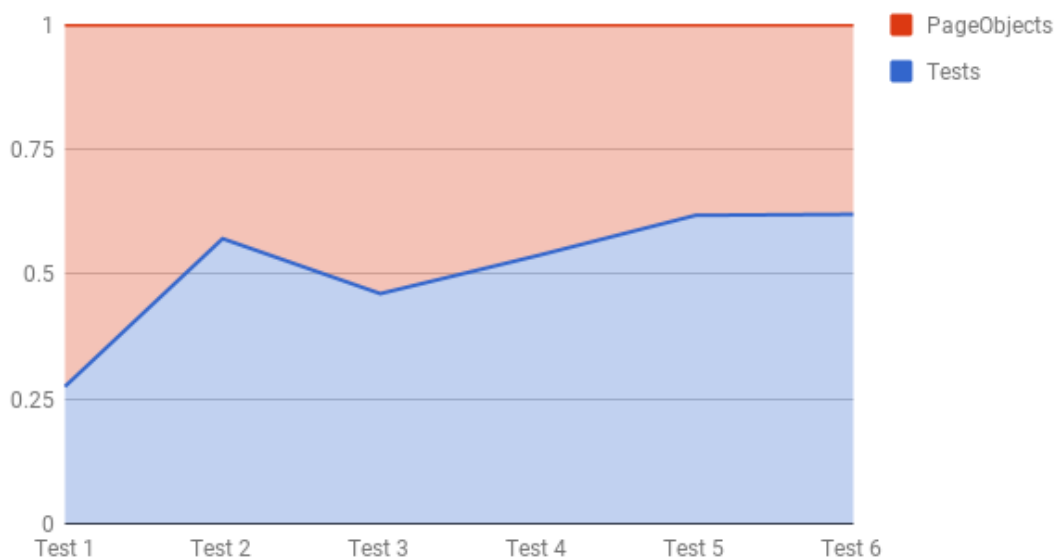


Figure 4.1: Graph over the difference in new LoC specific to the test and LoC added to the common PageObjects. The X-axis is sorted on the order the tests were implemented. Note how the reuse of PageObjects allow for a higher percentage of the work to be placed on the parts unique to the test itself.

The implementation time of the EyeAutomate test suite is shown in Table 4.2 where it is split into two categories: time until the test passed and time until the test felt robust. Which test needed the most improvement time was very context dependent, e.g. verifying cells in information tables could lead to a lot of errors.

Table 4.2: The implementation time, the size of the tests in lines of code (LoC) and size on disk for the EyeAutomate tests. The implementation time is split into the time it took for the tests to pass and the improvement time in order to reduce false positives and improve accuracy. LoC and file size are not used in this study, but was included as a base of comparison between other studies in the field. The test are in order of implementation.

EyeAutomate	Impl. time (seconds)	Improv. time (seconds)	LoC	File size (KB) Additions
Test 1	8672	12106	188	63.6
Test 2	406	783	10	3
Test 3	7661	0	201	45.5
Test 6	11523	1708	99	34
Test 5	11034	0	211	41.6
Test 4	10428	7341	229	52.4
Total	49724	21938	938	240.1
Average	8287	3656	156.3	40

EyeAutomate tests were significantly faster to implement than Selenium tests with the Selenium test suite taking 91% longer to implement. Looking at the individual test cases in Table 4.3 only one test takes longer to implement in EyeAutomate, and that is only by 15 minutes. In total, EyeAutomate tests took roughly 20 hours to implement while Selenium took 38 hours. Using a two-tailed T-test for two dependant means it was determined that the difference was statistically significant with an alpha of 0.05. The reason that a T-test for two dependent means was chosen is that the data was paired, the same test cases were implemented in two different tools.

Table 4.3: Table over the differences in implementation time between the tools. The p-value was determined with a two-tailed T-test for two dependent means.

Implementation differences	Selenium time (min)	EyeAutomate impl. + improv. time (min)	
Test 1	695.8	346.3	
Test 2	53	19.8	
Test 3	419.4	127.7	
Test 4	398.3	296.2	
Test 5	512.7	183.9	
Test 6	205.3	220.5	
Total	2284.9	1194.4	p = 0.039255
Average	380.8	199.1	

4.2 Maintenance Cost

The maintenance cost for the tools has been gathered using data from the historical testing. The stepping results can be seen in Table 4.4 and Table 4.5 for Selenium and EyeAutomate respectively. Concerning the total amount of time spent on maintaining tests, more time was spent on maintaining the EyeAutomate tests than the Selenium tests. The difference was 11 hours and 22 minutes for EyeAutomate compared to 7 hours and 47 minutes for Selenium, a difference of 32%. The average maintenance time for a single step was roughly 7 minutes for Selenium and around 10 minutes for EyeAutomate. The median time for each tool was 0 seconds. This is due to the majority of the samples the tests passing without the need of maintenance. For both tools, the majority of the time was spent on repairing broken tests. For the total repair time, there is a considerable difference between the tools, with the EyeAutomate tests in total taking more than twice as long to repair.

For handling bugs, the median time was zero for the test suites. The reason for this is due to the way found bugs are handled. If a bug is found from a test and it is still appearing during later runs of the same test, it would be flawed to measure the time it takes to "find" the bug again. Since the prior knowledge of the bug's existence makes it easier to find, the time it would take to find it again would be close to a few seconds. Consequently, the time was sampled as 0 for those instances.

Table 4.4: The stepping result for Selenium during the 65 steps. Occasional crashes was unknown errors causing the tests to fail, but passed after a re-run. The first five categories have their average, median and standard deviation based solely on their occurrences, i.e. The average of ‘Analysing broken tests’ is the total for the category divided by 19. The calculations for ‘Total per step’ is instead based on the full period of 65 steps.

Selenium	Total time spent (s)	Steps present	Average time (s)	Median time (s)	SD σ (s)
Analysing broken tests	5475	19	288.2	203	381.1
Repairing broken tests	14831	19	780.6	344	875.8
Handling found bugs	2181	24	90.9	0	264.7
Handling false negatives	3399	2	1699.5	1699.5	2106.5
Occasional crashes	2166	4	541.5	347	494.7
Total per step	28052	28/65	431.6	0	997.8

Table 4.5: The stepping result for EyeAutomate during the 65 steps. Occasional crashes was unknown errors causing the tests to fail, but passed after a re-run. The first five categories have their average, median and standard deviation based solely on their occurrences, i.e. the average of ‘Analysing broken tests’ is the total for the category divided by 22. The calculations for ‘Total per step’ is instead based on the full period of 65 steps.

EyeAutomate	Total time spent (s)	Steps present	Average time (s)	Median time (s)	SD σ (s)
Analysing broken tests	4041	22	183.7	131.5	158.8
Repairing broken tests	34243	22	1556.5	939	1609.8
Handling found bugs	1827	30	60.9	0	178
Handling false negatives	614	2	307	307	278.6
Occasional crashes	243	2	121.5	121.5	108.2
Total per step	40968	26/65	630.3	0	1310.2

The spread of the samples gathered during the historical can be seen in the boxplot in Figure 4.2. The box plot depicts eight boxes, categorised according to maintenance type and tool. Several outliers can be seen for category C. As described earlier, this is due to the way repeated bugs were measured. One category was excluded and is missing from the plot and maintenance calculations, the time it took to handle false negatives. The category was excluded due to its extremely low sample size, only two samples for both tools. False negatives would not normally

4. Results

be handled since they can only be found through cross-comparing different tools; it is normally too expensive for practitioners to maintain several test suites testing the same cases.

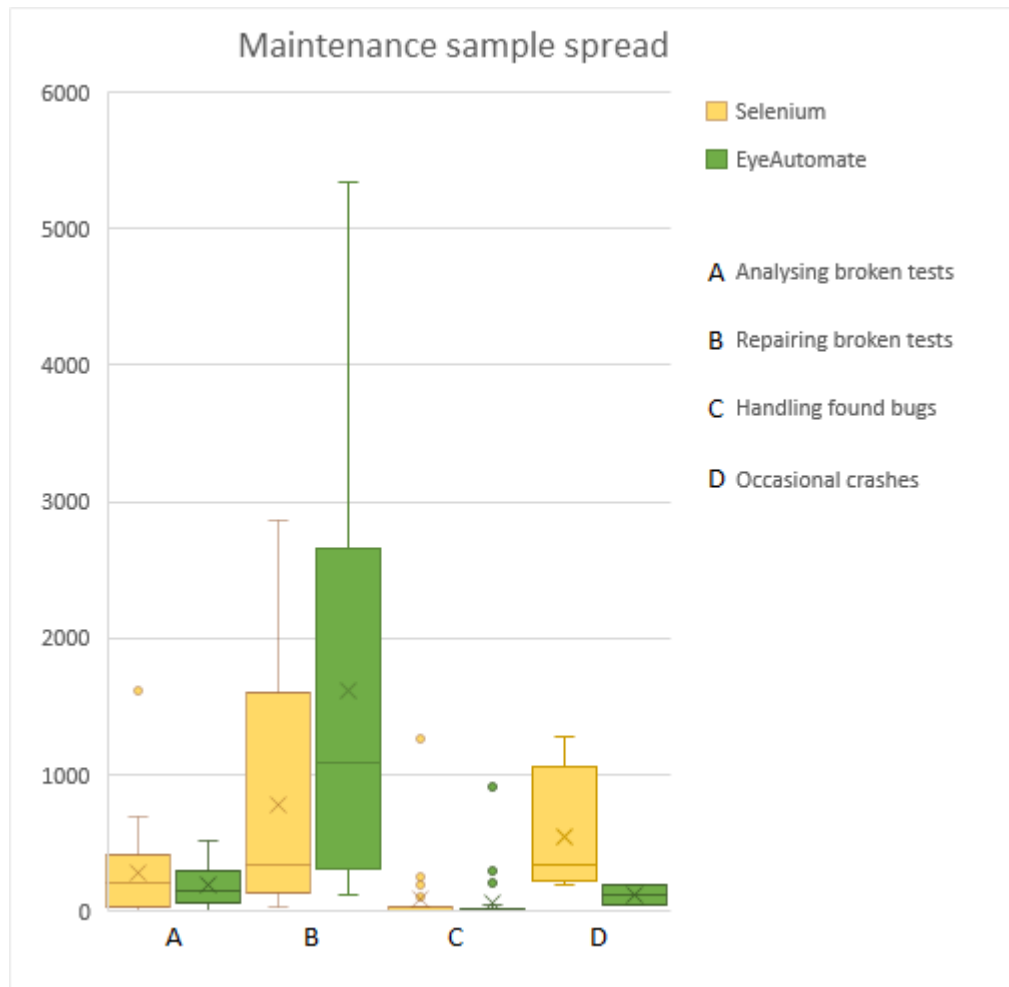


Figure 4.2: Boxplot describing the samples gathered from the historical testing. Note that group D consists of few data points, four data points for Selenium and two data points for EyeAutomate.

A graph showing time spent maintaining the tests can be seen in Figure 4.3. Here, the Y-axis displays the time in seconds spent on handling and maintaining a test suite for a given tool. The X-axis indicates the steps in which the tests were run. As seen in the graph, most of the maintenance occurred early during the stepping. During the latter half of the stepping, neither test suite needed much maintenance.

The spikes in the graph represent tests where significant time was spent on repairing the tests, most of these spikes can be explained. The first spike seen between step index 1 to 7 was mainly caused by the tests still being immature, manifesting as timing issues for Selenium and misclicks for EyeAutomate. The second spike seen from step index 7 to 15 were caused by a graphical overhaul of the CANEA system. Considerable time had to be spent on updating the locators for the Selenium and EyeAutomate test suites. The spike at step index 31 was caused by changes to different types of GUI inputs in the system such as date, text and drop-downs. The last spike for Selenium seen around step index 47 was due to a timing issue where a page transitioned too fast for the test. What these spikes add up to can be seen in Figure 4.4.

A histogram of the combined time it takes to analyse and repair a broken test is seen in Figure 4.5. The histogram depicts two samples of data: Selenium with a sample size of 43 and EyeAutomate with a sample size of 56. The median values for Selenium and EyeAutomate were 188 and 433 respectively. None of the sampled data seems to be normally distributed based on the shape of the histogram. A Shapiro-Wilk test confirmed that they were not normally distributed.

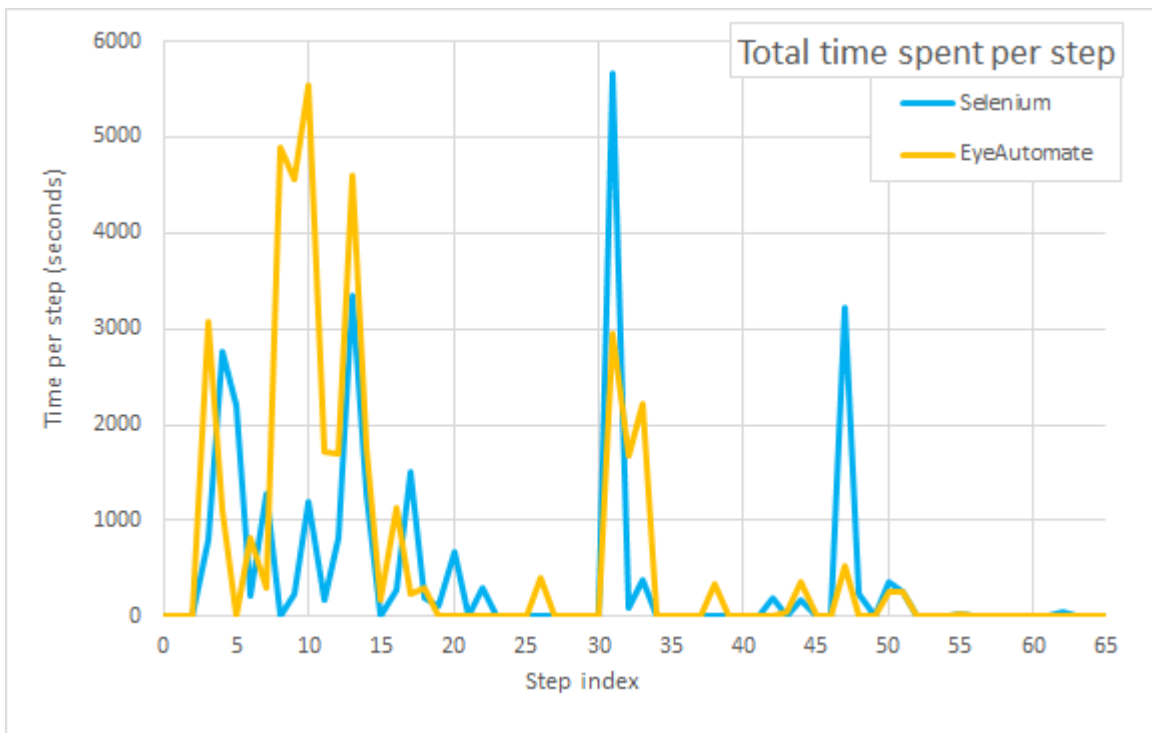


Figure 4.3: This graph shows how much time was spent on handling, analysing and repairing the tests totalled for each step. Note that step 1-7 and 59-65 only have one day between them while all the other steps have one week.

4. Results

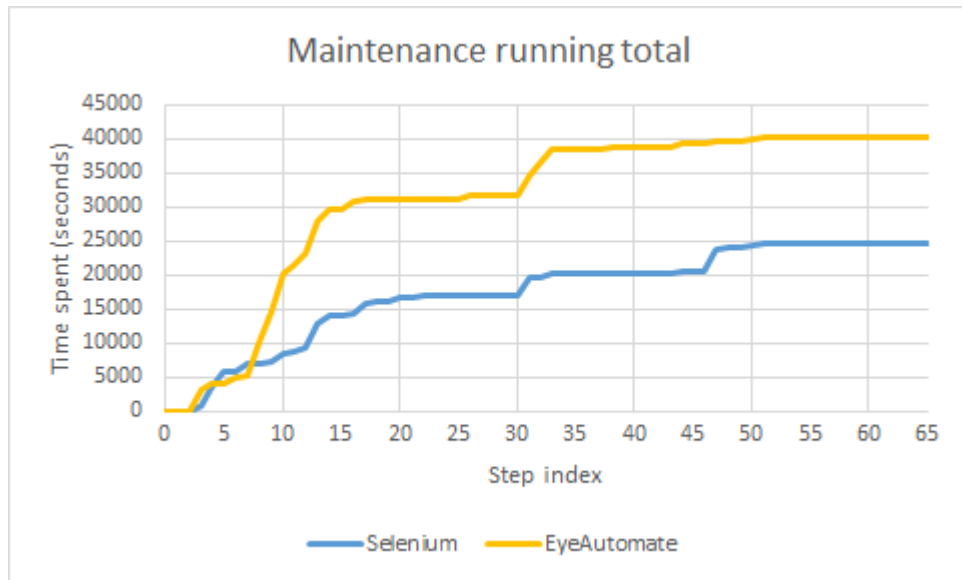


Figure 4.4: The running total of time spent handling the test per step.

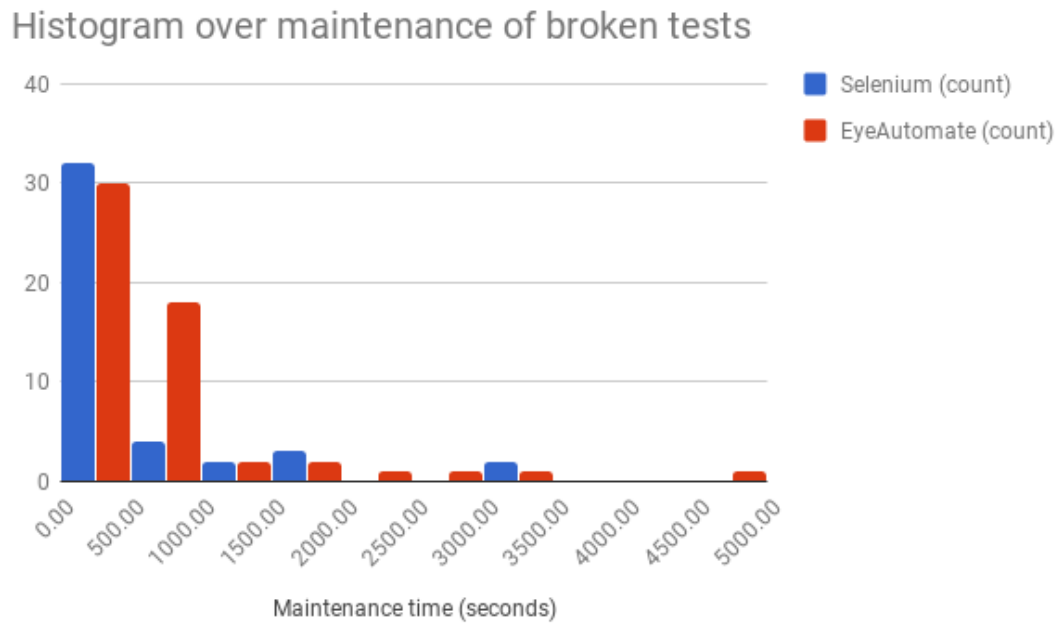


Figure 4.5: Histogram showing the distribution of the repair time for the tools.

Determining whether the two groups are significantly different through using Student's T-test would not be suitable, as the two samples are not normally distributed. An alternative would be to use the Mann-Whitney U test. This non-parametric test assumes that the samples are independent of each other and that the data is ordinal. Testing the two samples using a two-tailed Mann-Whitney U test concluded that the difference between the median of the two samples is statistically significant with a p-value of 0.00279159 at a significance level of 0.05.

Distributions of the maintenance cost spread evenly over set periods of time can be seen in Table 4.6. The values are derived from the total cost bar the time for handling false negatives as seen in Table 4.4 and Table 4.5.

Table 4.6: The maintenance cost over time. The numbers are the totals from Table 4.4 and Table 4.5 excluding the time for handling false negatives. The whole year is based on the actual data while the other yearly costs are extrapolated from a sub-set of the data. Monthly is the yearly time divided by 12 and daily is the yearly time divided by 226, the average number of workdays per year for an employee in Sweden.

Maintenance	Time period	Selenium time (s)	EyeAutomate time (s)
Based on first third (Extrapolated)	Yearly	52 453	93 855
	Monthly	4 288	7 821
	Daily	228	415
Based on second third (Extrapolated)	Yearly	8904	22 776
	Monthly	742	1 898
	Daily	39	101
Based on last third (Extrapolated)	Yearly	13 602	4 431
	Monthly	1 134	369
	Daily	60	20
Whole year (Measured)	Yearly	24 653	40 354
	Monthly	2 065	3 363
	Daily	109	179
Ratio		6.13:10	

4.3 Return on Investment

The base for return on investment, in this case, is the time spent on manual testing compared to the time spent implementing and maintaining an automated test suite. The time it takes to manually perform each test can be seen in Table 4.7. The data is the time it took for the testers at CANEA to perform the manual regression tests which the automated tests are based on. Normally these manual tests are run once before every release, which is every third month.

Table 4.7: The manual test suite execution time for use in return on investment. Selenium and EyeAutomate times are included for comparison and are averages from full-pass runs.

Execution time	Manual exec. (s)	Selenium exec. (s)	EyeAutomate exec. (s)
Total	4459	450	1812
Average	743.2	75	302

The total implementation cost was 20 hours for EyeAutomate and 38 hours for Selenium as seen in section 4.1. This is the cost that will have to be recuperated by running the tests since automatic tests don't need human supervision and therefore saves time compared to manual ones.

The maintenance cost will be combined with the implementation cost. There will be a continuous cost associated with repairing the automatic tests. In contrast, manual tests do not carry repair costs. The maintenance cost is analysed in section 4.2 and any repairs need to be included when comparing the investment to manual testing.

4.3.1 Calculated

Going by the average maintenance cost for the whole year in section 4.2, both Selenium and EyeAutomate have a higher maintenance cost than manual execution time, meaning that a return on investment will never be reached. Even looking at the best cases of the extrapolated sub-sets, second third for Selenium and last third for EyeAutomate, it would take 15 years and 5 years respectively to reach a return on investment. If any large changes to the system would happen during these years, the time to reach return on investment could rise even more.

By taking the yearly average and defining the maintenance cost as linear, return on investment can be estimated as seen in Table 4.8. Note that this average is based on weekly maintenance. This means that the maintenance cost would likely be marginally lower for the release and monthly estimate while being somewhat higher for the daily estimate.

Table 4.8: Calculated return on investment based on the implementation costs in Table 4.3 and the linear yearly averages from Table 4.6 and Table 4.7.

Return on investment	Selenium time	EyeAutomate time
1 per release (4/year)	Never	Never
Monthly	4.75 years	5.45 years
Weekly	35 weeks	20 weeks
Daily	32 days	17 days

4.3.2 Infrequent Runs

Normally the manual tests are run every three months. Plotting this together with the implementation and maintenance cost creates Figure 4.6. The figure assumes that the manual testing cost stays constant per run. Going by the visuals alone it is possible that a return on investment is reached after several years, but it is hard to be more specific without further investigation.

To get a better estimate of when a return on investment occurs in the long term, a trend-line prediction based on the first year was created. The best fit for both Selenium and EyeAutomate was a logarithmic curve and the resulting prediction can be seen in Figure 4.7. Due to the long time until a return on investment, even a few spikes of maintenance can disrupt this prediction quite a bit.

4. Results

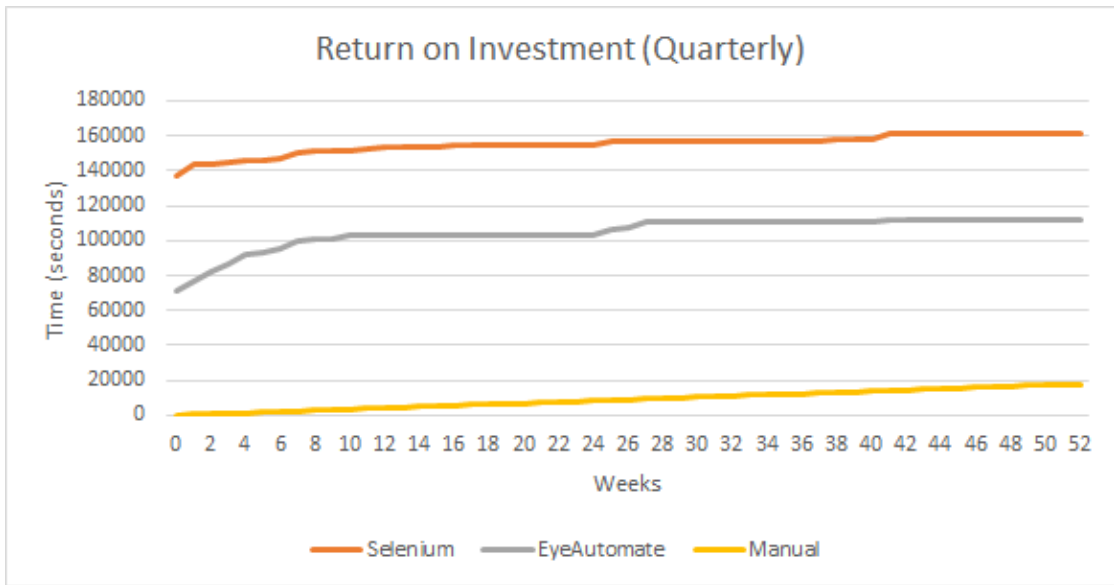


Figure 4.6: The manual test suite run quarterly compared to the the test suite investment for each tool.

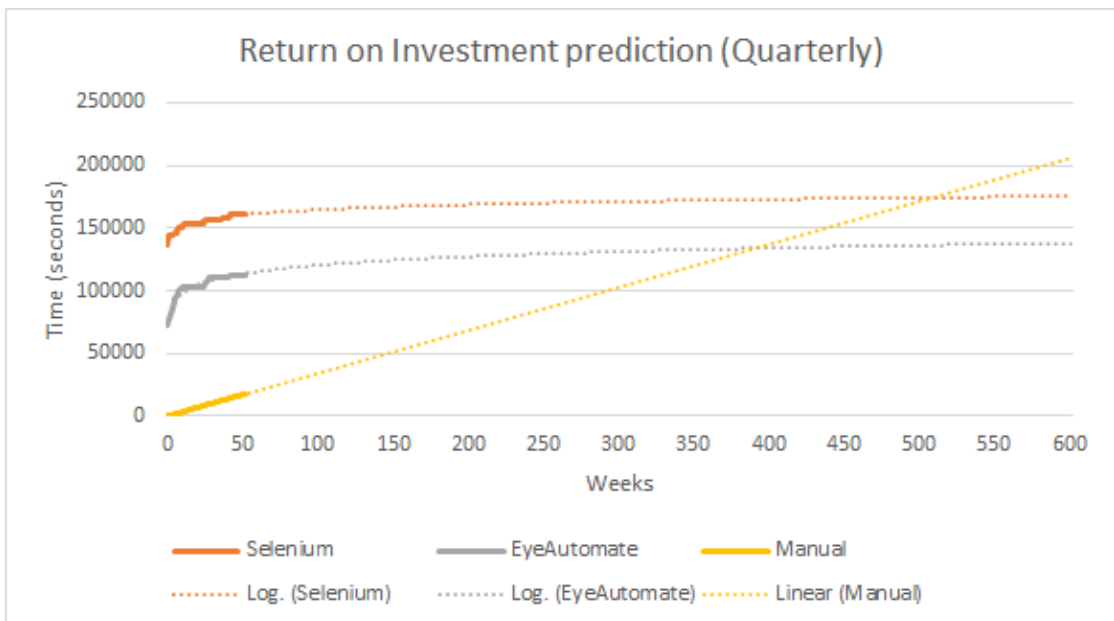


Figure 4.7: Prediction over time with quarterly test runs. Both Selenium and EyeAutomate had the best fit with a logarithmic function based on the first year. Selenium with $6045\ln(x) + 136818$ and EyeAutomate with $9919.8\ln(x) + 74676$. The manual cost was defined as a linear function $343x$ (a weekly average of 4459 every third month). Selenium equals the manual cost at 510 weeks, whereas EyeAutomate at 392 weeks.

4.3.3 Frequent Runs

The previous graph was based on quarterly runs. However, the purpose of the automated GUI tests was not be run quarterly. The automated tests were meant to be run frequently as regression tests, in order to find defects earlier. A graph plotted for weekly tests can be seen in Figure 4.8. In this case, a return on investment would be reached in 24 weeks for EyeAutomate and 36 weeks for Selenium.

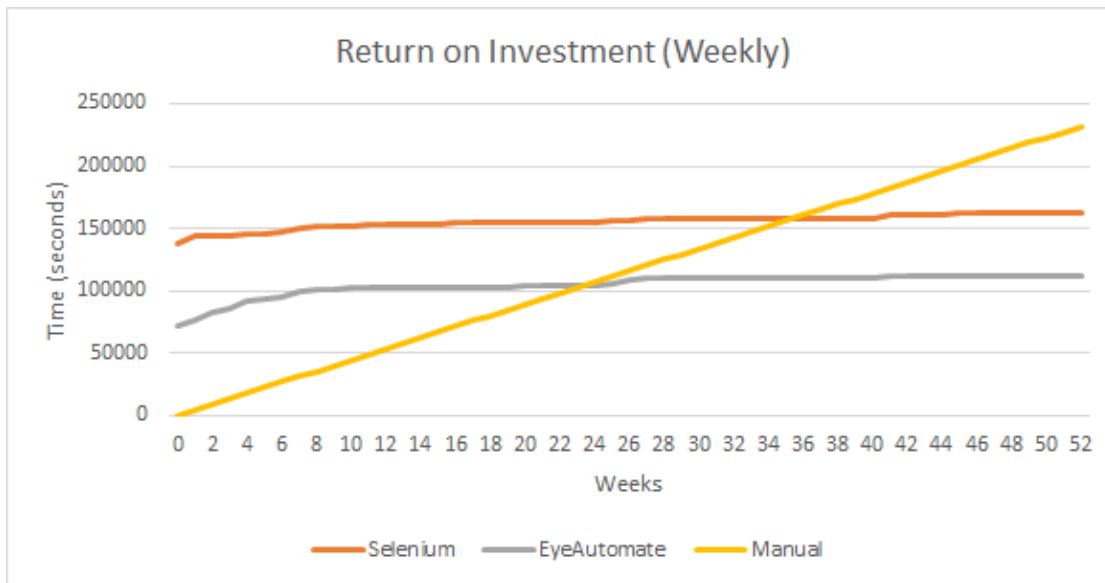


Figure 4.8: The manual test suite run weekly compared to the test suite investment for each tool. EyeAutomate crosses manual testing at 24 weeks, Selenium at 36 weeks.

4.4 Fault Detection Capabilities

The fault detecting capabilities of the tools have been explored using data gathered from the historical and artificial testing. This data is summarised in Table 4.9 and Table 4.10.

Table 4.9 was created from the data gathered during the historical testing. It shows all instances where the test suites of the tools would break or find a bug due to a change in the system. For Selenium, the total instances where the tests broke and needed repair was 28. For EyeAutomate, this number was 37. For all the

4. Results

changes to the system, 8% of them broke both test suites simultaneously. Using a Chi-Square test gives a P-value of 0.889 regarding the difference in how often the tests break between the tools, hinting at the tools being similar in how often they break but not with any statistically significant certainty.

Table 4.9: This table shows the number of unique cases where a test broke or a bug was found. The amount is shown in relation to the other tool, meaning a number "A (B)" shows that A findings were unique to the tool while B findings were present in both tools.

Unique findings	Selenium		EyeAutomate	
	Broken	Bugs found	Broken	Bugs found
Total (x) is common	23 (5)	1 (14)	32 (5)	4 (14)

The number of bugs detected by the Selenium tests was 15. For EyeAutomate, this number was 18. Most of the bugs found during the historical testing were shared between the tools with a total of 14 bugs. Based on the data gathered from the historical testing, 73% of all bugs were found by both tools. A Chi-Square test gives a P-value of 0.283 regarding the difference in bugs found between the tools, showing neither a significant difference nor a significant similarity. The unique bug found by Selenium was the absence of a drop down component which the corresponding EyeAutomate test missed. The four unique bugs found by the EyeAutomate test suite were graphical and layout related bugs.

The fault detecting capabilities of the tools were also examined with the artificial testing. The results gathered from this experiment are seen in Table 4.10. In regards to whether the tools follow the desired test outcome, the tools are split evenly with 12 successes and 7 failures. A success means that the executed test had the same result as the desired test outcome, whereas a failure means that the test did not achieve the same result. In the following paragraphs, the differences between the tools will be explained for each category of changes.

Application: Both Selenium and EyeAutomate succeeded in detecting the internal error and long sleep statement. Where the tools differed was when there were several sleep statements injected in the code, causing the Selenium test to pass while the EyeAutomate test failed. The reason for why the EyeAutomate test failed was because the injected sleep statements caused a timing issue during a page transition. With the extra delay added, the EyeAutomate script erroneously clicked on a GUI element not part of the test.

Browser: Almost all of the tests failed in some way when running them on a browser other than Chrome, the exception being Selenium in Internet Explorer. For Selenium, the reason why most test runs failed was due to timing issues and DOM-elements obscuring target elements, since the site became responsive before having fully loaded. Internet Explorer did not have these issues as it was slower to respond than Firefox and Edge, giving it time to load completely. For EyeAutomate, the main reason for why all test runs failed was because of the slightly different way each browser rendered the system.

Graphical: The graphical changes gave some different results. Both tools failed to detect that a GUI component was too big. Why EyeAutomate failed to find the larger component was because it falsely verified and identified another GUI component as the target. As for Selenium, the test does not validate the appearance of the GUI at all. For the rest of the changes, the EyeAutomate test was able to successfully distinguish between the defects and minor changes. While the Selenium did not manage to catch the differences with the text or images of a component, it did manage to successfully break when not finding a hidden element.

Layout: Both of the tools manage the first three changes without any difficulties: adding or removing a component not targeted by the test and hiding one that is. However, the tools were not able to detect when a component changed location significantly.

Structural: For the structural changes, the tools failed on different changes. Selenium failed when the tag type of a DOM-element was changed. EyeAutomate did not manage to fail when a targeted element was removed from the DOM. In this case, it identified another element as the targeted one and proceeded the test without failure.

4. Results

Table 4.10: The results from the artificial method. Check marks represent when the outcome matches the desired one.

Change	Desired Test Outcome	Selenium	EyeAutomate
Application			
Internal error	Fail	Fail (✓)	Fail (✓)
Long sleep	Fail	Fail (✓)	Fail (✓)
Shorter sleeps	Pass	Pass (✓)	Fail
Browser			
Edge	Pass	Fail	Fail
Firefox	Pass	Fail	Fail
Internet Explorer	Pass	Pass (✓)	Fail
Graphical			
Major size change	Fail	Pass	Pass
Minor size change	Pass	Pass (✓)	Pass (✓)
Text resource	Fail	Pass	Fail (✓)
Image resource	Fail	Pass	Fail (✓)
Hide element	Fail	Fail (✓)	Fail (✓)
Layout			
Add	Pass	Pass (✓)	Pass (✓)
Remove	Pass	Pass (✓)	Pass (✓)
Hide	Fail	Fail (✓)	Fail (✓)
Change location	Fail	Pass	Pass
Structural			
Change tag	Pass	Fail	Pass (✓)
Surround tag	Pass	Pass (✓)	Pass (✓)
Move up	Pass	Pass (✓)	Pass (✓)
Remove	Fail	Fail (✓)	Pass

4.5 Differences Between the Tools

This section compiles the quantitative results from the research methods in order to specify differences between the tools.

As seen in Table 4.3, there is a difference in implementation cost between the tools. In five out of six cases, implementing a manual test case in EyeAutomate was faster than in Selenium. Running a ‘two-tailed T-test for two dependant means’ concluded the difference to be statistically significant.

The data shown in Figure 4.5 indicates that there is a difference in maintenance cost between the test suites. A two-tailed Mann-Whitney U test concluded that the samples are statistically different, with the Selenium test suite having a lower maintenance cost than the EyeAutomate one. The maintenance costs for the test suites were not consistent over the year. As seen in Figure 4.3 and Table 4.6, most maintenance occurred during the first third of the year and then gradually decreasing over time.

The total cost of implementing and maintaining a test suite for a given tool can be compared to the time it takes to perform the same test suite manually. The calculated return on investment is seen in Table 4.8. As seen in the table, depending on how often the manual test suite would be run affects the time to a positive return on investment. Comparing the tools, the EyeAutomate test suite reaches a positive return on investment slightly faster than the Selenium test suite.

As seen in Table 4.9, most bugs found during the historical testing were found by both test suites. The EyeAutomate test suite did manage to find more unique bugs with four compared to one of Selenium. The artificial testing revealed the tolerance of the test suites concerning system and environment changes. Table 4.10 shows the results of the artificial changes. The test suites were equal in the number of successes and failures. The data seen both from the historical and artificial testing suggests that the tools are sensitive to some types of changes.

4.6 Qualitative data

This section contains a summarised version of the data gathered from the interviews described in section 3.4. The full result from the interviews can be read in Appendix A.

Note that while EyeAutomate is the program actually running the scripts, EyeStudio is the IDE used during the interviews. Therefore, only the term EyeStudio was used to reduce confusion. This mismatch in terminology does not pose a problem since EyeStudio can only write EyeAutomate scripts, making it easy to extract whether a comment is related to EyeAutomate or EyeStudio. Any comments regarding the syntax or running of the scripts relate to EyeAutomate while any comments regarding the IDE relate to EyeStudio.

Prior experience with the tools:

The interviewees had very limited experience with automated GUI testing in general. Only one of them had used a GUI testing tool before, and then only briefly for performance testing. Of the five people interviewed, four were developers and one was a tester. It was chosen to not only have developers in order to get more varied opinions. None had any experience with EyeStudio but all the developers had experience in C# and Visual Studio, which were used for the Selenium tests.

Impressions of Selenium:

Everyone interviewed mentioned to a lesser or greater extent that programming knowledge is required to use Selenium. The developers were generally positive and found Selenium easy to use due to its familiar language and development environment, the tester claimed it was: "Not for me".

The PageObject pattern is meant to increase the reusability of the Selenium test suite. True to the intention, interviewees felt that PageObjects made it easier to reuse code. Therefore, the use of PageObjects was seen as something positive; one interviewee even claiming PageObjects as one of the main reasons Selenium felt better than expected. A downside with the pattern brought up by a few interviewees was the possibility of the PageObjects becoming bloated as the test suite grew.

Selenium's reliance on DOM-based locators was only brought up a few times during the interviews. Most of the interviewees grasped the concept fairly quickly and did not even bring it up, apart from some syntactical questions while writing the tests.

Two interviewees claimed that the knowledge of using CSS to form locators in the DOM makes Selenium easier to use. Another saw the use of CSS as having the potential for a positive feedback loop in that the usage improves the knowledge of CSS in the team. This could lead to a higher product quality which in turn leads to fewer tests breaking.

Impressions of EyeStudio:

The interviewees found EyeStudio easy to use. The visual script with images made it easy to get an overview and the clickable instructions made it easy get started. One interviewee said that the program had an intuitive workflow which led it to feel natural to use, stating that it felt "a lot like recording yourself and just doing, rather than thinking".

Several interviewees said that a lot of technical knowledge is not required to get started in EyeStudio, mostly due to the intuitive workflow. Since there are menus with all the available commands, a beginner does not have to remember every command. They can instead look it up. However, there were some areas where experience is required, specifically when it came to handling timing issues and different recognition modes.

Even though the reception of EyeStudio was mostly positive, there were also quite a few negative remarks. Many interviewees noted how sensitive and error-prone the image recognition felt, how the test hijacks the computer while running and how difficult it is to reuse code. Some minor things brought up were: the lack of IntelliSense, the unintuitive names for images and the inability to run several tests in parallel.

Automated GUI tests replacing manual:

The general consensus of the interviewees is that an automated GUI test can replace a manual one, but not all of them. Preferably they would want the most boring and repetitive test automated as a complement to manual tests because a machine is faster and does not get bored. The machine is also more consistent and can perform the same steps in the exact same way every time. However, they feel that the tests would have to be well defined in order to be automatised, meaning that exploratory testing cannot be replaced. A human could find issues outside a strictly defined test case by following intuition and testing related features. The interviewees also feel that an automated GUI test would lead to more false positives and false negatives, partly because a human can be more forgiving or reasonable about visual changes.

Viability for the company:

All the interviewees agree that automated GUI testing can be viable for the company. They believe that the automated GUI tests should be used as regression tests and be run at least daily in order to find bugs early. The two tools had one main point each brought up regarding viability: Selenium would be easy to integrate with the current environment, and EyeStudio could be written by a wider range of personnel. These two points also lined up with what the interviewees felt

was best for the company. Some felt that the robustness and ease of integration of Selenium should be utilised by having the developers implement the tests. Others felt that it would be better to free up the developers by having another department implement the tests in EyeStudio.

Maintenance:

While most interviewees were fine with repairing tests due expected changes, how much repairing they felt was sustainable varied greatly. Some thought that half of what they currently spent on manual testing would be reasonable, which would total about 60 hours over three months. Others felt that no more than five minutes every other week is acceptable, totalling 30 minutes over three months. Selenium was said to be easier to maintain, at least for developers. EyeStudio gave the impression that it would break a lot.

Impression of automated GUI testing:

Before trying to write the tests, each interviewee stated their expectations of automated GUI testing. Some expressed expectations of a high maintenance cost. Others stated the high difficulty of implementing automated GUI testing correctly. This leads to automatic GUI testing being somewhat of a "holy grail", very valuable but hard to get right.

After the having tried both tools, the same question was asked again to see if any opinions have changed. Two interviewees said that generally their expectations were met. Most of the interviewees were more positive towards Selenium after having tried it. EyeStudio was claimed to be easy enough that anyone with some computer knowledge could use it. Finally, there was a comment that it is important for the company to think about who is going to write the test.

5

Discussion

This chapter will go through the implications, weaknesses and significance of this study. The research methods and the results in regards to the research questions are under discussion. The limitations and threats to validity to this study are discussed as well. Future research suggestions based on the results of this study concludes this chapter.

5.1 Research Methods

Three different research methods were used during this study. Why they were chosen can be seen in section 3.1. This section discusses the outcome of the methods, what worked and what did not.

5.1.1 Historical

The most time-consuming part of the historical phase was the stepping between versions, which often went slower than anticipated. In general the stepping went well, but there were two main issues slowing the process down. The first was the large size of the product, making it take around 15 minutes to just get the next version and rebuild the project. The second issue was old dependencies or other technical troubles, requiring workarounds to get the project working. Add the time to run and repair the tests and the total time spent averaged about three hours per step.

Even though the method might have taken longer to perform than desirable, it did produce good and seemingly realistic data. The method is mimicking the execution

of an automatic GUI test suite on a fixed interval and maintaining the tests as the product changes, which is how such tests would realistically be implemented in a company. The best would be to have this fixed interval set to daily, but due to time constraints it was set to one week for this study. Something which might have improved the results would be to pick a random day each week instead of always picking the same. By randomising the day, the chance of missing issues introduced and fixed between samples are up to chance rather than a systematic choice in the study.

In practice, there is two-way relationship between the development of the system and the automated tests, where one influences the other in a potential feedback-loop [27]. The historical method only captured how the development of the system influenced the automated tests, not the other way around. Determining how the automated GUI tests influences the development of the system would be possible to capture through an observational study.

5.1.2 Artificial

The artificial experiment tested the defect finding capabilities of the tools through manually injected changes. An advantage with this method compared to the historical testing is that it allows for a wider range of changes to the system than what would normally occur during development.

A major factor to the result of the artificial experiment is the amount of human decision making needed. One such decision is deciding what type of changes should be injected. In the experiment, a combination of research findings, the experiences of the company and the researchers' experiences when working with the tools were used to choose changes. A risk concerning the aforementioned sampling strategies is that they are sensitive to bias.

Capturing and defining all relevant changes has also been a challenge in the experiment. Depending on how the changes are categorised and defined, one tool could be favoured over another. Injecting a change to the system is not always fully apparent. In the experiment, a locator-based approach with randomisation was used. For most of the changes, the approach worked well enough.

Deciding how severe a change would be has been a challenge in the artificial experiment. Whereas for some changes, the change can be clearly defined such as removing an element from the DOM. For other changes, the degree of change has to be decided, such as moving a GUI component from its original position or increas-

ing the size of a component. Randomisation was used when deciding this degree. However, for some changes such as moving a GUI component, the randomisation tactic was not suitable due to the many degrees of freedom the changes have. A manual ad hoc strategy was employed instead where the change was implemented to the degree that it looked like a severe fault in the system.

Due to the various factors mentioned above, it is not certain that the results would be the same if the experiment would be repeated. An improvement to the experiment design would be to inject and test a change more than one time in the system, using a different degree of changes and injection places for each one. This repeated approach would yield more data. Furthermore, the result would be more robust to the shortcomings described above. The reason why this approach was not used was due to time constraints.

5.1.3 Interviews

Interviews were performed in order to capture qualitative data regarding the testing tools. Since the personnel at the company had no experience with the tools of this study, the interviews were designed to teach the interviewee about the tools before moving onto the actual interview. This presented a large risk to introduce bias among the interviewees since the researchers were behind not only the teaching, but the questions and evaluation as well. This threat to validity made it harder to define what and how to teach each interviewee, as discussed in section 5.7.

The test and walk-through used to teach the interviewees were designed to test the essentials of both tools and how to use them in a test suite. While the test was on the right track, the pilot interview helped to focus on and which parts needed to be expanded or removed. The resulting test and walk-through seemed to be appreciated by the interviewees, they felt that the walk-through prepared them well and the test covered both basic and challenging scenarios. The non-developer, however, needed a lot of help during the Selenium test.

5.2 Maintenance Cost

The associated cost for maintaining an automated test suite is the subject of RQ2. This section will compare the maintenance cost between the tools, go through differences and describe the implications of the results.

5.2.1 Repair Cost differences

The most time-consuming task when maintaining the test suites has been repairing tests. Between the tools, the EyeAutomate test suite had a higher repair cost. One possible explanation for the higher repair cost for EyeAutomate could be that each repair instance is verified by rerunning the test, which adds time to the EyeAutomate test suite since it has a noticeably slower execution time than the Selenium test suite, as seen in Table 4.7. The average EyeAutomate execution was 302 seconds per test. This longer execution time might also explain the notable difference between samples in the 500-1000 seconds bucket in the histogram shown in Figure 4.5, with 16 instances of EyeAutomate maintenance compared to 4 Selenium instances.

Another potential factor causing the difference in repair time could be the PageObject pattern used in the Selenium test suite. The PageObject pattern has been studied before with results pointing towards a decrease in maintenance cost for a test suite [10]. The historical testing had no control group to compare the PageObject test with, which in this case would be an equivalent Selenium test suite without using the PageObject pattern. Consequently, there can be no conclusive remarks on how much of the difference between Selenium and EyeAutomate was due to the PageObject pattern.

5.2.2 Fluctuating Maintenance Cost

Both tools have seen inconsistent maintenance cost over the period of testing. As seen in Figure 4.3, most maintenance occurred during the first third of the year. One explanation for this is that the system underwent a lot of graphical changes during this period, affecting both the image- and DOM-locators. Another contributing factor could be that there is a maturation effect where the tests need to be gradually improved before becoming more resilient to acceptable changes. This maturation effect means that more training with the tools prior to implementation would reduce this first robustness improvement cost. There could also be a potential learning effect with the researchers becoming more adept with the tools as time passes, decreasing the time it takes to repair a test.

In the historical testing, the test suites were run with weekly intervals between versions, except for two specific weeks where a daily sampling strategy was used. Opting for two weeks with daily sampling was done to verify if important data was missed by stepping weekly. The daily sampling was placed at the start and

end of the year in order to see differences between a new and mature test suite. More daily samples would have been taken during the year if time had allowed it. With a more frequent sampling strategy, such as running the test suites daily for the whole year, the measured maintenance cost would most likely be somewhat higher.

5.2.3 Conclusion

With regards to maintenance cost, the Selenium test suite had, in general, a lower maintenance cost than the EyeAutomate one; Selenium having a 32% lower maintenance cost than EyeAutomate. However, both tools appear to taper off with regards to maintenance when looking at Figure 4.4 and the prediction in Figure 4.7. While an automated test suite always needs to be updated when changes are made to areas affected by tests, over time the stability has hopefully increased to the point that it only breaks from deliberate changes and found defects.

5.3 Return on Investment

Return on investment is one of the main foci of this study, both from an industrial viewpoint and in the amount of data it uses. Return on investment is the entirety of research question RQ3, but it also uses the results from the maintenance examination RQ2, together with both implementation cost and manual test execution time.

5.3.1 Implementation Cost

The tests took a long time to implement, partly due to design. As seen in Table 4.3 it took more than three hours per test case on average. One reason for this is that the test cases were based on manual ones, often very long and testing multiple features in a sequence, something that is normally avoided in automatic tests. Another reason is the attempt to slow down the learning effect by switching between the tools often in order to get a fair comparison between them. In the end, this will lead to a slower return on investment, which means the return on investment calculations, seen in section 4.3, are most likely on the conservative side.

5.3.2 Maintenance Cost

The maintenance discussed in the previous section 5.2 can be split into three categories: average-, actual- and predicted cost.

The average maintenance cost shown in Table 4.6 can be used to quickly get a decent prediction for return on investment. This was done in Table 4.8. However, this kind of prediction does not take any trends into consideration.

When looking at the actual maintenance shown in Figure 4.3 it is apparent that the maintenance is not linear, but rather happens in spikes. There seems to be a spike in maintenance cost right after implementing the test suite. In this case, it was due to the newly implemented tests lacking robustness. This robustness maintenance is more visible in Figure 4.4 where about one and a half hours were spent on the first seven steps for both tools. Since these first seven steps were part of a sample week, it means that the first week accounted for 20% and 13% of the yearly total for Selenium and EyeAutomate respectively. Using the actual maintenance instead of the average, return on investment can be reached in 23-36 weeks as seen in Figure 4.8 if the test suites are run weekly. However, since the manual would not actually be run weekly because of the cost to do so, it could be argued that the more frequent runs only bring an increase in product quality and not a faster return on investment. If the tests were run quarterly the return on investment would be reach in what appears to be several years as seen in Figure 4.6.

When looking at several years in the future, as when testing is done quarterly, a predictive model is needed. Figure 4.7 shows a logarithmic prediction based on the maintenance of the first year, resulting in a return on investment after 7.5 years at best. However, maintenance costs are very context sensitive and often come as spikes as mentioned before, especially if any more graphical overhauls occur like the one between week 2 and 8. Considering the slowly rising logarithmic prediction together with the risks of overhauls over a long time, it is safe to say that 7.5 years is a very optimistic prediction.

5.3.3 Qualitative

Estimating return on investment was not the focus of interviews. However, suggestions were gathered on how to best reach it, both regarding how often to run the tests and who should write or maintain them.

Every person interviewed agreed that the automated tests should be run at least daily in order to be worthwhile. This is because the tests would detect bugs earlier than the infrequent manual tests. As discussed previously, the more frequent the test suite is run, the faster it meets a return on investment.

Another point brought up that affects the return on investment is who writes and maintains the tests. The interviewees mentioned having different departments responsible for the tasks. This can heavily affect the maintenance cost since the tools have different knowledge requirements. If a tool is given to people without the proper training, the time to return on investment can increase significantly.

5.3.4 Conclusion

The tools Selenium and EyeAutomate have similar performance when it comes to return on investment with EyeAutomate being slightly better due to the lower implementation cost. However, the tools are very different in their use and how much prior knowledge is required. It is therefore important to consider who will be writing the tests and chose which tool to use based on that. Chosen correctly, return on investment will be met within a year if the test suite is run fairly frequently. If run daily, it could be met within a month.

5.4 Fault Detection Capabilities

To answer SRQ4, the fault detection capabilities of the tools have been evaluated in this study. How the tools fared in finding defects will be discussed in this section.

5.4.1 Test Scenario

The manual tests, which the automated tests are based on, are mainly concerned with regression testing functionality of the system. Much like any other kind of tests, the tests are limited in what defects they will find by the test scenario. Performing this study with automated test suites based on other scenarios would likely reveal other defects.

5.4.2 Comparison

Concerning the number of bugs found from the historical testing, the EyeAutomate test suite found three more bugs than Selenium. These bugs were graphical and layout bugs. The EyeAutomate test suite carries both advantages and disadvantages with its sensitivity to visual changes. The results from the artificial method, seen in Table 4.10, showcase some of these changes such as changing icon or text. For both the historical and the artificial experiment, there were instances where the EyeAutomate test suite failed at determining the absence of GUI components. The test would instead select another GUI component and continue with the test. In the historical testing, 73% of all detected bugs were found by both tools. So in a context similar to the historical, the fault detection capabilities of the tools are quite similar. Whether or not this holds for any other kind of software projects has not been examined.

5.4.3 Conclusion

The main difference, in regards to the fault detection capabilities between the tools, is for finding graphical bugs. Here, EyeAutomate is more capable than Selenium. However, EyeAutomate's sensitivity to graphical changes can also be a hindrance, with the tool having a higher tendency of finding the wrong locator and continuing the test than Selenium, which was observed in both the historical and artificial testing.

5.5 Selenium vs. EyeAutomate Observations

While many of the previous sections include comparisons between the tools EyeAutomate and Selenium and build towards RQ1, this section is solely dedicated to it. The researchers have taken note of many differences between the tools while using them throughout the study. Some of the more considerable ones are discussed in this section.

5.5.1 Test Implementation

Both the test suites were quite heavy to implement when compared to the manual cost of doing the tests. With the average time to implement a test being at least 15 times that of the average time to perform it manually, the implementation time is far from negligible.

Using EyeAutomate, the main issue slowing down the implementation time is the time spent preventing false negatives. When an EyeAutomate test first passes, it is robust to small changes. However, this robustness also includes changes to important details, meaning that the test does not fail even if it should. This is discussed further in subsection 5.5.5.

The process of implementing Selenium tests was in general slower than the EyeAutomate one, since it involved programming and looking through the DOM instead of just selecting images. There was some upfront work required in Selenium to set up the tests and handling the connection to the browser, but most of the upfront cost came from creating PageObjects.

5.5.2 PageObjects

Using the PageObject pattern brought both positives and negatives with its lower maintenance but higher upfront cost. The PageObjects required a large effort upfront because they represent the GUI of the website. Because when committed to PageObjects, there can be no testing of the GUI without them. Another minor issue with using the PageObject pattern is the unavoidable context switching while working on a specific test only to pause and work on a generic PageObject. On the other hand, when a base of PageObjects has been created they instead lower implementation cost, as PageObjects can quickly be reused between tests.

The maintenance feels easier when using PageObjects, especially if multiple related tests fail at the same time. As mentioned in subsection 5.2.1 it cannot be said how much of the difference in maintenance cost between Selenium and EyeAutomate can be attributed to the PageObject pattern. However, both the researchers and several of the experienced interviewees agree that the PageObject pattern feels natural to use together with Selenium.

5.5.3 Prior Knowledge

The tools differ a lot when it comes who can use them. For example, an opinion that was repeated during the interviews when talking about Selenium was "experience is required". Since the tools are constantly being compared, this view refers both to the need of a programming background in order to effectively use Selenium and to how EyeAutomate does not need one. EyeAutomate is easy to get into and most people can get productive with the tool fairly quickly, as long as they have some general experience with computers and manual testing.

That said, both tools have some quirks that an experienced user knows how to avoid. Selenium suffers from timing issues and elements not being accessible when expected. EyeAutomate can solve the majority of the timing related issues with a step delay but has its own problems with uniquely identifying images and manipulating components without unique features. Many of these issues are things a new user encounters a few times, spends a lot of time fixing and then learns to recognise them in advance next time.

5.5.4 Locators

The locators used is the biggest difference between the tools, with both kinds having their own benefits and disadvantages. Selenium uses DOM-based locators while EyeAutomate uses image-based locators and how they differ will be discussed in this section. How the locators work on a higher level can be read more about in subsection 2.1.4 and subsection 2.1.5.

EyeAutomate is great for quickly creating functioning locators through its screen-capture tool. A problem is that the tool can be too lax in its image comparison. This can lead to obviously different images being recognised as equal. This comparison error can often be solved by changing the recognition mode to a more strict one, or by increasing the size of the image to form a more unique identifier. The disadvantage of these solutions is the added risk that the image is not recognised at all. It is therefore important to balance how much the locator image should contain and the recognition mode to use with it.

Selenium is very precise in its choice of locators, which causes problems for dynamically loaded components or elements without unique locators. Elements without unique selectors on the element itself has to be located by using the surrounding structure, leading to locators sensitive to structural changes. Dynamically loaded

components are often auto-generated, which not only cause them to have poor locators, they are also very sensitive to timing issues because the test has to wait for everything to load correctly. While not as much trial and error as EyeAutomate, these issues are also a balancing act in order to get a unique locator without making the locator too sensitive to changes in surrounding components.

5.5.5 Data Verification

It is often important during GUI tests that specific details are present and correct, something that the different tools are not equally adept at handling. Selenium cannot verify the graphical look of components but has no problem fetching text strings for verifying details. EyeAutomate on the other hand primarily verifies using images. Image recognition can cause problems when verifying text or handling similar looking components. Verifying text in EyeAutomate was often solved by marking text with the cursor, copying it and then using the clipboard for verification. Similar images or identical images were especially troublesome for information tables. Here, imaginative workarounds had to be used in order to locate specific cells.

Since EyeAutomate has dynamic loading of Java programs, it is sometimes worthwhile to create custom commands with Selenium integration in order to solve the more advanced instances of data verification. Creating custom commands, with or without Selenium integration, is something that is supported and encouraged by the creators of EyeAutomate, as stated on the EyeAutomate homepage¹. Although, this does place a significantly higher demand on the users' programming knowledge.

¹<http://eyeautomate.com/resources/EyeAutomateExpertCourse.pdf> (Accessed 2018-06-12)

5.5.6 Conclusion

Selenium and EyeAutomate are two very different tools. Their purposes are the same but the distinct locator technologies and writing styles result in two tools for two different target audiences. Both tools are sufficient for testing with some compromises when compared to each other. Selenium is more robust with its element-based locators, with the trade-off being insensitivity to visual changes. EyeAutomate is written in its own scripting language using the accessible IDE EyeStudio, with the trade-off being the lack of functionality a more established IDE and language can provide.

5.6 Limitations

The work on this study has been limited in various ways. A number of these limitations have been identified. These are related to the tools studied, the historical stepping and the test case definition.

This study was limited to only two tools: Selenium and EyeAutomate. While more automated GUI test tools could have been included in the study, they were excluded due to time constraints. Using a tool each from the 2nd and 3rd generation was deemed sufficient for the purpose of this study.

Another limitation in the study has been the stepping length in the historical research method. A shorter, more frequent stepping would have been possible and would yield more data. Why a longer stepping length of one week was used as a default was mainly motivated by the amount of work a shorter stepping length would bring. Even if a shorter stepping length would have been used, the stepping period for the historical would most likely have to be shorter than a year due to the limited time frame of the study.

Defining a test case has not been included in any of the measurements defined in this study. As the test cases had already been defined before the study as manual test cases, this factor was excluded from this study.

5.7 Threats to Validity

The results found in this study are subject to validity threats. This section will go through the identified threats and how they were handled. The threats were categorised as: threats to conclusion validity, threats to internal validity, threats to construct validity and threats to external validity.

5.7.1 Conclusion Validity

A number of different outcomes have been observed during this study, this section goes through said outcomes and analyses them in regards to how likely it is that an error has occurred.

Implementation cost comparison:

The implementation cost difference was found significant in Table 4.3. It is therefore possible that a Type I error has been made. Considering only six tests were written in each tool, there is an argument that the small sample size could have caused an error. Leotta et al. also performed a comparison study between 2nd- and 3rd generation tools and came to the opposite conclusion. They found Selenium to be faster. However, their way of implementing the tests for the 3rd generation tool differed greatly to how it was done in this study.

However, a study by Leotta et al. comes to similar results [5]. The study compares Selenium WebDriver, simply called Selenium in this study, with Selenium IDE, a capture-replay version of Selenium very similar to how EyeAutomate was used in this study. Worth to mention is that the same authors published another study where they compared Selenium WebDriver to the 3rd generation tool Sikuli API. In their study, Selenium was found to be faster to implement than Sikuli API [3], but it has a low relevance to this study since the implementation of Sikuli API tests have very little to do with how EyeAutomate tests are written.

Maintenance cost comparison:

Just like with the implementation cost, the maintenance cost difference was also found to be significant, meaning there is a risk that a Type I error has been made. There is a huge variance in the data as seen in Figure 4.2, meaning that there is a risk of error. A similar study came to the result that Selenium WebDriver cost less to maintain than the capture-replay tool Selenium IDE [5].

Manual and automatic execution time:

The manual execution time is based on one run of the manual test suite, the majority being performed by a very experienced manual tester. This makes it unlikely that the time would decrease with more samples from other people, but if it did, it would affect the return on investment of this study for the worse. A higher manual execution time would improve the return on investment, but the change would likely be marginal with only a few weeks or days change for the monthly and daily run respectively. The automatic execution time is of little interest since it is run without supervision. If a major change to the automatic execution time occurs, it could affect the maintenance time or how often the tests can be run. The observer effect, the effect that subjects performs differently when under observation, should also be taken into account when evaluating the measured manual execution time.

Unique bugs and breaks:

No statistically significant results could be found from the data in Table 4.9, neither for bugs nor for breaks. The study would have benefited from a significant result either as being different or being the same, since a conclusion for the number of bugs and breaks between the tool is a good point of comparison. Although it might be caused by bias, the experiences of the researchers points towards it being very likely that there is a difference between the tools. With a larger sample size, this difference will likely appear.

Fault detection capabilities:

Very similar to the previous subject of unique bugs and breaks. No significance was found but the differences were distinct enough that a larger sample size would likely show a significant difference between the tools.

Interviews:

There is a possibility that important nuances were lost during the analysing step because the researchers selected which answers to focus on. To prevent this, great care were taken to include all sides in the summaries about the interviews. Furthermore, all the answers to the interviews can be seen in Appendix A together with how many expressed the opinions.

5.7.2 Internal validity

Whether or not the treatment caused the measured outcome is the subject of internal validity. In this study, there may be other factors which could have influenced the outcome of the methods.

Biases:

The human aspect and cognitive biases are factors which could have influenced the results. The researchers' involvement in designing the tests, repairing tests, defining defects and setting up the interviews are some procedures where personal bias could influence the results of the study. The researchers have been sponsored by the company behind EyeAutomate, Auqtus AB, through trial EyeAutomate licenses. The researchers have also received technical support from Auqtus AB. The researchers' thesis supervisor is also involved in Auqtus AB as a board member. It has been the researchers' intention to keep these procedures away from personal influences as much as possible. When dealing with decisions, the researchers aimed to face the issue from the same point of view as the case company, which is an objective point of view to gather whether the usage of any of the tools would be beneficial.

Learning effect:

One potential factor which could have influenced the results is the learning effect. With time, a test developer could be more experienced with creating automated GUI tests. This would influence implementation cost and maintenance cost. Similarly, the initial experience level of a test developer can affect implementation cost and maintenance cost for each tool. In order to reduce the impact of the learning effect, tasks were split in a way so that the tools were used evenly by the researchers. However, since the maintenance of the historical testing is a year compressed into a few weeks, the implementations of the tests were still fresh in the researchers' memories which most likely improved the repairing process.

5.7.3 Construct validity

Construct validity concerns whether or not the research methods have captured relevant data to answer the research questions.

The historical testing has mainly used time as a measurement in order to answer the research questions regarding maintenance cost and return on investment. Using time as a measure of value is probably in line with many companies. However, something the research methods miss to capture is the cost of other aspects such as hardware costs, potential licensing costs, and the time it took to get comfortable with the tools. These factors should be taken into account when evaluating the results.

The interview method is subject to construct validity threats. Bias could have influenced the interview questions in favour of one tool over the other. So, in order to alleviate these concerns the researchers used the same questions for both tools.

5.7.4 External validity

Whether or not the results can be generalised outside the scope of this study is the concern of external validity.

The results of this study are limited to the tools Selenium and EyeAutomate. The results should therefore be interpreted with these tools in mind. Similarly, this study used data based on a single system. As such, caution should be exercised when applying the results to other systems of different sizes and types.

5.8 Contributions

This study adds to the body of knowledge in many different ways, with several aspects concerning automated GUI testing being examined in an industrial context. Long-term usage of Selenium and EyeAutomate has been simulated using version control history, something which, to the researchers' knowledge, has not been attempted in an automated GUI testing study before.

The study fills a research gap concerning the maintenance cost of a 3rd generation test suite as it grows during software evolution, a gap which has been noted in previous studies [33, 36, 43].

The results of this study can also be of use to practitioners considering adopting automated GUI testing. This thesis provides information about the associated costs and benefits for 2nd- and 3rd generation tools, therefore providing a basis for a decision.

5.9 Future work

This thesis has investigated various aspects of automated GUI testing. A pattern noticed during this study is that the maintenance cost for an automated GUI test suite is not consistent but can have spikes. An interesting aspect to evaluate in the future is the maintenance cost of an automated GUI test suite for a system under frequent changes. Investigating strategies to mitigate maintenance cost during those development periods could be valuable for both researchers and organisations. Another potential future research topic is the use of automated GUI tests in conjunction with manual tests in order to determine the strengths and weaknesses of both approaches.

6

Conclusion

The purpose of this study was to compare two very different GUI testing tools with a heavy focus on maintenance. The industry moves more and more towards automation. GUI testing is something that has previously been viewed as strictly manual labour. This creates a demand from the industry to get information on what can be expected from automated GUI testing, how to implement it and the expected return on investment. When considering contemporary image-based GUI testing tools, there is also a need from academia to see how it performs compared to more established ones.

This study was performed with the support of interviews and a small experiment, but the main method used to reach the research goals was a simulation of automated regression test suites using earlier system changes. The tools under inspection were the new image-based tool EyeAutomate and the established element-based tool Selenium.

The results differed between the tools depending on the aspect, especially in their usability. Selenium tests took 91% longer to implement but had 32% lower maintenance cost than EyeAutomate. Considering the manual testing cost, both tools would reach a return on investment within one year if run weekly and within weeks if run daily. However, while the tools are somewhat similar in return on investment, they are very different to use. The qualitative examination saw a large difference in who could use the tools efficiently, with people without a programming background being able to use EyeAutomate but not Selenium.

One of the main things to take away from this study is that different tools are needed for different people and types of products, something that is often overlooked when comparing testing tools. With this in mind, it is easy to make automated GUI testing beneficial as long as the product is not undergoing constant graphical overhauls.

Bibliography

- [1] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving GUI-directed test scripts. In *Proceedings - International Conference on Software Engineering*, 2009. ISBN 9781424434527. doi: 10.1109/ICSE.2009.5070540.
- [2] Atif M. Memon. Gui testing: Pitfalls and process. *Computer*, 2002. ISSN 00189162. doi: 10.1109/MC.2002.1023795.
- [3] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Visual vs. DOM-based web locators: An empirical study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014. ISSN 03029743. doi: 10.1007/978-3-319-08245-5{_}19.
- [4] Miikka Kuutila, Mika Mäntylä, and Päivi Raulamo-Jurvanen. Benchmarking Web-testing-Selenium versus Watir and the Choice of Programming Language and Browser. *arXiv preprint arXiv:1611.00578*, 2016.
- [5] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 272–281, 2013. ISBN 9781479929313. doi: 10.1109/WCRE.2013.6671302.
- [6] Dudekula Mohammad Rafi, Katam Reddy, Kiran Moses, and Kai Petersen. Benefits and Limitations of Automated Software Testing : Systematic Literature Review and Practitioner Survey. *Automation of Software Test (AST), 2012 7th International Workshop on*, 2012. doi: 10.1109/IWAST.2012.6228988.
- [7] E Alégroth and R Feldt. On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering*, 2017. ISSN 13823256 (ISSN). doi: 10.1007/s10664-016-9497-6.

- [8] Emil Alégroth, Robert Feldt, and Pirjo Kolström. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology*, 73:66–80, 2016. ISSN 09505849. doi: 10.1016/j.infsof.2016.01.012.
- [9] Pekka Aho, Matias Suarez, Atif Memon, and Teemu Kanstren. Making GUI Testing Practical: Bridging the Gaps. In *Proceedings - 12th International Conference on Information Technology: New Generations, ITNG 2015*, 2015. ISBN 9781479988273. doi: 10.1109/ITNG.2015.77.
- [10] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013*, 2013. ISBN 978-0-7695-4993-4. doi: 10.1109/ICSTW.2013.19.
- [11] Richard Potter. Triggers: Guiding Automation with Pixels to Achieve Data Access. In Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors, *Watch What I Do: Programming by Demonstration*, chapter 17, pages 361–380. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-03213-9. URL <http://dl.acm.org/citation.cfm?id=168080.168129>.
- [12] Luke S Zettlemoyer, Robert St. Amant, and Martin S Dulberg. IBOTS: Agent Control Through the User Interface. In *Proceedings of the 4th International Conference on Intelligent User Interfaces, IUI '99*, pages 31–37, New York, NY, USA, 1999. ACM. ISBN 1-58113-098-8. doi: 10.1145/291080.291087. URL <http://doi.acm.org/10.1145/291080.291087>.
- [13] Ellis Horowitz and Zafar Singhera. Graphical user interface testing. *Technical eport Us C-C S-93-5*, 4(8), 1993.
- [14] Emil Borjesson. Industrial applicability of visual GUI testing for system and acceptance test automation. In *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, 2012. ISBN 9780769546704. doi: 10.1109/ICST.2012.129.
- [15] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. Visual GUI testing in practice: challenges, problems and limitations. *Empirical Software Engineering*, 20(3):694–744, 2015. doi: 10.1007/s10664-013-9293-5.
- [16] Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *Proceedings - AGILE Conference, 2006*, 2006. ISBN 0769525628. doi: 10.1109/AGILE.2006.19.

-
- [17] Andrea Adamoli, Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Automated GUI performance testing. *Software Quality Journal*, 2011. ISSN 15731367. doi: 10.1007/s11219-011-9135-x.
- [18] Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Annals of Software Engineering*, 2002. ISSN 10227091. doi: 10.1023/A:1020549507418.
- [19] Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. A case study-based comparison of web testing techniques applied to AJAX web applications. In *International Journal on Software Tools for Technology Transfer*, 2008. ISBN 1000900800. doi: 10.1007/s10009-008-0086-x.
- [20] Atif M Memon and Mary Lou Soffa. Regression testing of GUIs. *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE '03*, 2003. ISSN 01635948. doi: 10.1145/940071.940088.
- [21] Atif Memon, Adithya Nagarajan, and Qing Xie. Automating regression testing for evolving GUI software. In *Journal of Software Maintenance and Evolution*, 2005. ISBN 1532-060X. doi: 10.1002/smr.305.
- [22] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology*, 2008. ISSN 1049331X. doi: 10.1145/1416563.1416564.
- [23] A. Michail. Helping users avoid bugs in GUI applications. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005. ISBN 1-59593-963-2. doi: 10.1109/ICSE.2005.1553553.
- [24] Vahid Garousi and Mika V. Mäntylä. A systematic literature review of literature reviews in software testing, 2016. ISSN 09505849.
- [25] Vahid Garousi and Michael Felderer. Worlds Apart: Industrial and Academic Focus Areas in Software Testing. *IEEE Software*, 2017. ISSN 07407459. doi: 10.1109/MS.2017.3641116.
- [26] V. Garousi, M. Felderer, M. Kuhrmann, and K. Herkiloğlu. What industry wants from Academia in software testing? Hearing practitioners' opinions. In *ACM International Conference Proceeding Series*, 2017. ISBN 9781450348041. doi: 10.1145/3084226.3084264.
- [27] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international*

- conference on Software engineering - ICSE '05*, 2005. ISBN 1595939632. doi: 10.1145/1062455.1062556.
- [28] Päivi Raulamo-Jurvanen, Mika Mäntylä, and Vahid Garousi. Choosing the Right Test Automation Tool: A Grey Literature Review of Practitioner Sources. In *EASE*17*, pages 21–30, Karlskrona, Sweden, 2017. ACM. doi: 10.1145/3084226.3084252.
- [29] Erik Sjösten-Andersson and Lars Pareto. Costs and Benefits of Structure-aware Capture/Replay tools. In *SERPS'06*, Umeå, Sweden, 2006.
- [30] Emil Alégroth. *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*. PhD thesis, Chalmers University of Technology and Göteborg University, Göteborg, 2015. URL <https://research.chalmers.se/publication/221145>.
- [31] Yuan-Fang Li, Paramjit K Das, and David L Dowe. Two decades of Web application testing - A survey of recent advances. *Information Systems*, 2014. ISSN 0306-4379. doi: <http://dx.doi.org.pc124152.oulu.fi:8080/10.1016/j.is.2014.02.001>.
- [32] Päivi Raulamo-Jurvanen, Kari Kakkonen, and Mika Mäntylä. Using surveys and Web-scraping to select tools for software testing consultancy. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016. ISBN 9783319490939. doi: 10.1007/978-3-319-49094-6{_}18.
- [33] Emil Alégroth, Robert Feldt, and Helena H. Olsson. Transitioning manual system test suites to automated testing: An industrial case study. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, 2013. ISBN 978-0-7695-4968-2. doi: 10.1109/ICST.2013.14.
- [34] Emil Alegroth, Zebao Gao, Rafael Oliveira, and Atif Memon. Conceptualization and evaluation of component-based testing unified with visual GUI testing: An empirical study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, 2015. ISBN 9781479971251. doi: 10.1109/ICST.2015.7102584.
- [35] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Automated generation of visual web tests from DOM-based web tests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC '15*, 2015. ISBN 9781450331968. doi: 10.1145/2695664.2695847.
- [36] Emil Borjesson and Robert Feldt. Automated System Testing Using Vi-

- sual GUI Testing Tools: A Comparative Study in Industry. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 350–359, 2012. ISSN 9780769546704. doi: 10.1109/ICST.2012.115. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6200127>.
- [37] Vahid Garousi, Wasif Afzal, Adem Çağlar, İhsan Berk Işık, Berker Baydan, Seçkin Çaylak, Ahmet Zeki Boyraz, Burak Yolaçan, and Kadir Herkiloğlu. Comparing automated visual GUI testing tools: an industrial case study. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing - A-TEST 2017*, pages 21–28, 2017. ISBN 9781450351553. doi: 10.1145/3121245.3121250. URL <http://dl.acm.org/citation.cfm?doid=3121245.3121250>.
- [38] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 2005. ISSN 00985589. doi: 10.1109/TSE.2005.117.
- [39] Emil Alegroth, Johan Gustafsson, Henrik Ivarsson, and Robert Feldt. Replicating Rare Software Failures with Exploratory Visual GUI Testing. *IEEE Software*, 2017. ISSN 07407459. doi: 10.1109/MS.2017.3571568.
- [40] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. Maintenance of Android Widget-based GUI Testing: A Taxonomy of test case modification causes. In *ICSTW*, pages 151–158, 2018. doi: 10.1109/ICSTW.2018.00044.
- [41] Kai Petersen and Claes Wohlin. Context in industrial software engineering research. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, 2009. ISBN 9781424448418. doi: 10.1109/ESEM.2009.5316010.
- [42] Emil Alégroth and Robert Feldt. Industrial application of visual GUI testing: Lessons learned. In *Continuous software engineering*. Springer International Publishing, 2014. ISBN 9783319112831. doi: 10.1007/978-3-319-11283-1-11.
- [43] Emil Alégroth, Michel Nass, and Helena H. Olsson. JAutomate: A tool for system- and acceptance-test automation. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, 2013. ISBN 978-0-7695-4968-2. doi: 10.1109/ICST.2013.61.

A

Interview Answers

Table A.1: The questions in this table were asked before the interviewees tried the tools. The number of interviewees who expressed the same or similar opinion are stated within parenthesis on the left.

Interview answers — Pretest questions	
Experience with C#?	
(4)	Yes.
(1)	No.
Experience with Visual Studio?	
(4)	Yes.
(1)	Occasional usage for small tasks.
Experience with EyeStudio?	
(5)	None.
Experience with automated GUI testing?	
(2)	None.
(2)	Some but not much.
(1)	Has used an earlier tool, but only for performance testing.
Expectations of automated GUI testing before tests?	
(2)	Feels like it requires a lot of maintenance.
(2)	Is sort of a "holy grail", but it can be hard to get a return on investment due to the difficulty of implementing it correctly.
(1)	Sceptically optimistic.
(1)	Faster than a manual test.

A. Interview Answers

Table A.2: The questions in this table concerns the tool Selenium and were asked after the interviewees had tried both tools. The number of interviewees who expressed the same or similar opinion are stated within parenthesis on the left.

Interview answers — Selenium WebDriver questions	
What is good about the tool?	
(3)	Familiar structure, encourages reusability.
(3)	Familiar to programmers.
(1)	Many different ways to find things (elements).
(1)	Results in a positive feedback loop with CSS. Knowledge of CSS is needed to write the test, this knowledge improves the quality of the site, making it easier to test.
(1)	There is an expected behaviour to the tests, similar to programming.
(1)	Not as sensitive (less prone to break) as EyeStudio.
(1)	Faster execution speed.
(1)	The modularity allows for easier maintenance.
(1)	The tests are precise.
(1)	Feels more flexible (than EyeStudio).
(1)	Doesn't hijack the computer (like EyeStudio does).
What is bad about the tool? Potential problem?	
(3)	Programming experience is required.
(2)	PageObjects could grow into bloat, hard to get an overview.
(1)	Requires knowledge about locators.
(1)	There is an occasional need for Thread.Sleep() calls.
(1)	Tests are fragile to markup changes.
How is the ease of use of the tool?	
(2)	It's ok, especially after some time with the tool.
(1)	Pretty much the same as EyeStudio after the introduction.
(1)	Validating the absence of elements required some workarounds.
(1)	Easy to use, mostly due to the familiarity with the language. It is made even easier due to the use of the DOM through the browser.
(1)	"Not for me"
(1)	Knowledge of CSS selectors, JavaScript and NUnit makes Selenium easier to use.
(1)	Reusing parts becomes easier with the PageObject pattern.
Does the tool fit in the company's workflow?	
(3)	Easy to integrate into existing test suite.
(1)	Potential positive feedback loop with CSS knowledge.
(1)	Steps it (the testing standards of the company) up to the next level.
(1)	Adds value as regression tests.

(1)	Because it runs in the background it can use parallelisation to improve execution speeds.
(1)	Depends if it can be run in headless mode or not.
(1)	It needs to be run nightly.
(1)	Could replace existing automated GUI tests (Coded UI).
(1)	Could fit if done by developers, not really possible for QA.
Any other general impressions of the tool?	
(2)	Went better than expected, especially with the easy to understand code of the PageObjects.
(1)	Most of the features of Selenium are context-dependent whether they are good or not.
(1)	With an established language & IDE (C# & Visual studio) you have access to powerful refactoring tools.
(1)	It was hard.
(1)	Having precise error messages were good.
Can a Selenium test replace a manual test?	
(3)	Yes, at least most of them.
(2)	Wouldn't find bugs outside the test case.
(1)	Yes, by they would need to be carefully specified.
(1)	Yes, but unsure which types of tests.
(1)	Would give more false negatives.
(1)	Tired tester often just go by the book without reflection anyway.
(1)	Exploratory testing would find more bugs.

Table A.3: The questions in this table concerns the tool EyeStudio and were asked after the interviewees had tried both tools. The number of interviewees who expressed the same or similar opinion are stated within parenthesis on the left.

Interview answers — EyeStudio questions	
What is good about the tool?	
(3)	You don't need a lot of technical knowledge to get started. Due to images and ready, clickable, instructions.
(2)	It's easy to get an overview of the test.
(2)	Easy to use the basic instructions.
(2)	Didn't require writing much code.
(1)	Faster debugging (than Selenium).
(1)	There was a nice "flow" where it was more doing than thinking. It felt natural, a lot like simply recording yourself.
What is bad about the tool? Potential problem?	

A. Interview Answers

(3)	More error prone with sensitive images.
(2)	Hijacks the computer while running.
(2)	Low reusability of code.
(2)	Fine-tuning, and therefore knowledge is required for writing tests.
(1)	The autogenerated names for new images are not intuitive.
(1)	Checks all windows for matches, not just the desired/active one.
(1)	Slower than Selenium, especially with added delays.
(1)	Hard to get an overview if they (the tests) get too long.
(1)	Limited to the tool (EyeStudio), compared to the many languages supported by Selenium.
(1)	No intellisense.
(1)	Difficult to structure code.
(1)	Can find the "wrong" element leading to false positives/negatives.
(1)	Difficult to run tests in parallel.
How is the ease of use of the tool?	
(4)	Easy to understand and use.
(1)	More people in the organisation (in addition to developers) could use it.
(1)	The screenshot in the report were nice.
(1)	Working with an IDE with buttons and menus (for commands) make the development of tests easier.
(1)	It had some irritating moments, the (poorly working) automatic indentation in particular.
Does the tool fit in the company's workflow?	
(1)	Same as Selenium (for me).
(1)	EyeStudio is easier and can use more resources (not only developers) for writing tests.
(1)	Some of the repetitive tests, such as the smoke tests, could really benefit from being automated.
(1)	Best to run nightly.
(1)	Happier employees who doesn't have to run boring tests manually.
(1)	Return on investment is important, depending on the implementation cost it can be hard to break even.
(1)	Selenium fit better.
(1)	Would work, but depends on how often changes make the tests break.
(1)	If it is supported by the current framework (TeamCity).
Any other general impressions of the tool?	
(2)	Some experience is required to recognise when to switch recognition mode or knowing when timing could be a problem.

(2)	Easy to get into.
(1)	Fairly positive.
(1)	Minimal setup required.
Can an EyeAutomate test replace a manual test?	
(2)	Yes, most of them.
(1)	Yes, but they would need to be carefully specified.
(1)	It would miss details a human tester could find.
(1)	Unsure, feels like it breaks a lot.
(1)	Yes, but unsure which types of tests.

Table A.4: The questions in this table were asked at the end to round up the interview. The number of interviewees who expressed the same or similar opinion are stated within parenthesis on the left.

Interview answers — Posttest, automated GUI testing questions	
Is automated GUI testing valid for the company?	
(4)	Yes.
(2)	Can function as regression tests.
(1)	More enjoyable for developers to write a test than running it manually.
(1)	I think so.
How could it be implemented to greatest effect?	
(3)	Test suite should be run at least once a day.
(2)	Have another department (not developers) write the tests with EyeStudio while doing regular testing.
(1)	Selenium could be used while teaching developers about css.
(1)	An automated regression suite allows for faster finding of bugs.
(1)	Use strict test cases specified by QA and have the developers implement them in Selenium.
(1)	Use it as a complement to manual testing.
(1)	Have someone working on it (half- to full-time).
What is sustainable maintenance wise when it comes to automated GUI testing?	
(2)	About half of what is currently spent on manual testing per release. (= 20-30 hours/month)
(1)	1 test per day for a 5 minute fix. (= 1,75 hours/month)
(1)	Fails due to recent (expected) changes are not a problem to fix. Otherwise about 5 minutes every second week. (= 10 minutes/month)
(1)	Image changes are ok. Other breaks such as timing or browser related ones, 1-2 (5 minute) fixes over a 3 month period. (= 2,5 minutes/month)

A. Interview Answers

(1)	It would not be sustainable if a lot of time would have to be spent repairing a test suite at the end of each release period.
(1)	A selenium test would be more maintainable than an EyeStudio one for a programmer.
What can automated GUI testing do that a human can't do?	
(5)	Exact. A machine can repeat the same steps in the same way each time (without getting bored and skipping/missing steps).
(2)	Frequent tests allow for defects to be found earlier.
(2)	It is faster than a human.
(1)	Can be used as a proof of quality towards customer.
(1)	Frees human resources for other tasks.
What can't automated GUI testing do that human can?	
(4)	The machine cannot look outside the scope of it's test. Thereby missing apparent bugs/issues to the side.
(2)	A human can draw conclusions and intuitively test related areas not normally included in the test case.
(1)	A human can be more forgiving about minor changes, causing fewer false positives.
(1)	Have difficulty/cannot finding weird bugs or glitches, doesn't have a concept of what "looks" wrong.
Does the expectation on automated GUI testing differ after the tests?	
(3)	More positive towards Selenium.
(2)	Mostly what was expected.
(1)	EyeStudio is easy enough that anyone could write it.
(1)	It's important for the company to think about who should write the tests.

B

Selenium example

Figure B.1 shows three different types of locators. Link text is convenient to use, but there could be duplicates if there are many links on a page. CSS locators are very powerful and was the most commonly used locator in this study, the downside lies in their complexity, making them hard to use for the inexperienced. XPath have much of the same power that CSS has, but is often associated with absolute paths similar to the one displayed in the figure. While they have some functionality making them necessary, they are slower than CSS in most cases and should be avoided. Both CSS and XPath can be written as absolute paths, which are very fragile to changes and should be avoided.

B. Selenium example


Locating by link text

- Arts
- Biography
- Geography
- History
- Mathematics
- Science

```
<li style="position:absolute; left:0; top:0;">  
  <a href="/wiki/Portal:Arts" title="Portal:Arts">Arts</a>  
</li>
```

```
[FindsBy(How = How.LinkText, Using = "Arts")]  
private IWebElement artsLink;
```

Locating by CSS



Selenium is a suite of tools to automate web browsers across many platforms.

```
<div id="mBody">  
  <div id="sidebar">  
      
  </div>  
</div>
```

```
[FindsBy(How = How.CssSelector, Using = "div#sidebar img[alt='Selenium Logo']")]  
private IWebElement selLogo;
```

Locating by XPath

3.0	[16]	Active
17.5	[17]	
0.10.3	[18]	Active

```
<tr>...</tr>  
<tr>  
  <td>...</td>  
  <td>Windows</td>  
  <td>Windows, Web</td>  
  <td>...</td>  
  <td style="background: #dd" <td style="background: #9f9" <td>17.5</td>  
  <td>...</td> == $0  
</td></td>  
</tr>  
<tr>...</tr>  
<tr>...</tr>
```

```
[FindsBy(How = How.XPath, Using = "//*[@id='mw-content-text']/div/table[2]/tbody/tr[18]/td[8]/sup/a")]  
private IWebElement tableReference;
```

Figure B.1: An example of three different types of locators: link text, CSS and XPath.

C

EyeAutomate example

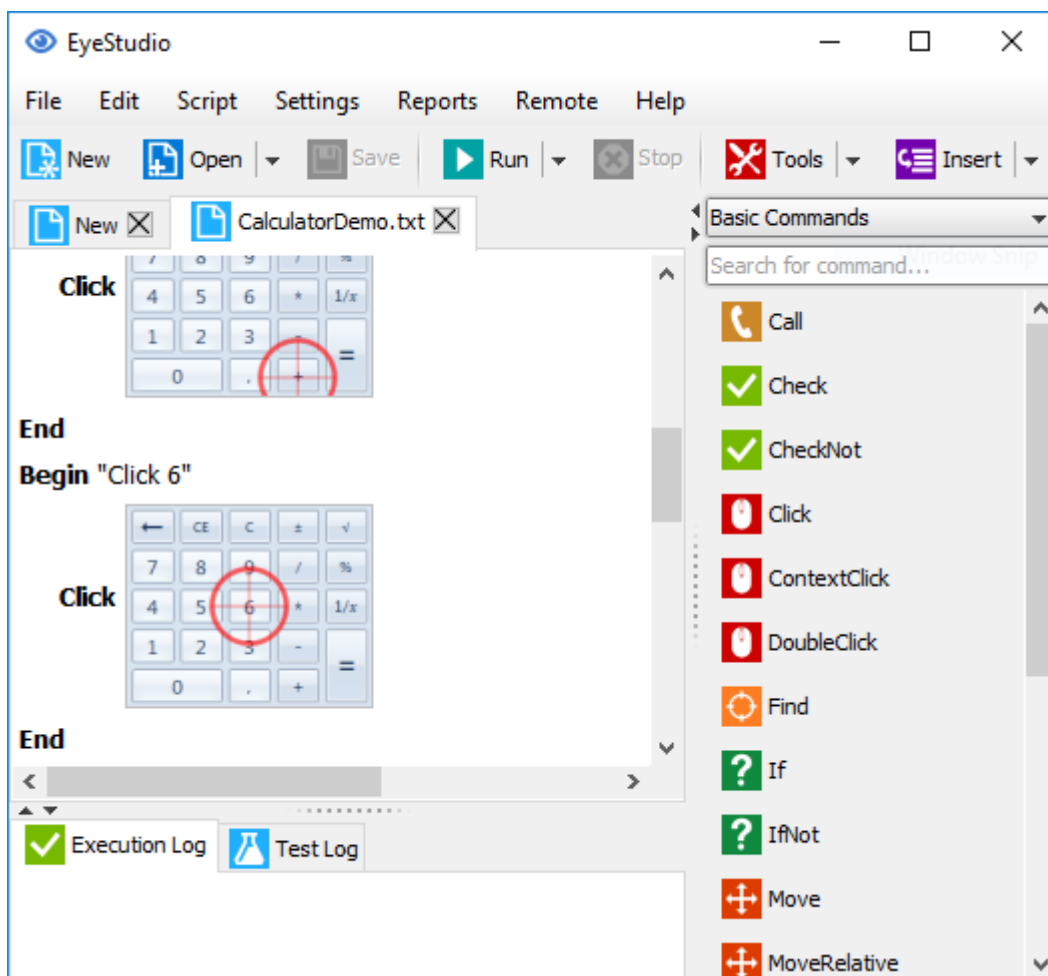


Figure C.1: EyeStudio with an EyeAutomate script.