IMPLEMENTING GUI TESTING USING WEBDRIVERIO TO COMPANY'S VUE.JS BASED WEB APPLICATION

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering and Digital Transformation, Master's Thesis

2022

Mikko Nummila

Examiners: Associate professor Jussi Kasurinen

M.Sc. (Tech.) Ville Nenonen

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Mikko Nummila

GUI testauksen implementointi WebdriverIO:lla yrityksen Vue.js pohjaiseen verkkosovellukseen

Tietotekniikan Diplomityö

2022

83 sivua, 3 kuvaa, 7 taulukkoa ja 4 liitettä

Tarkastajat: Apulaisprofessori Jussi Kasurinen ja Diplomi-insinööri Ville Nenonen

Avainsanat: Design science, GUI testing, testing automation, WebdriverIO, implementation, software testing

Tämä opinnäytetyö kuvaa WebdriverIO selainautomaatiotestikehysympäristön implementoinnin yrityksen Vue.js pohjaiseen verkkosovellukseen ja kehitysputkeen. Tätä varten verkkosovelluksen taustajärjestelmistä tehtiin malli Mock Service Workerillä. Tulokset ovat, että testiskriptejä voidaan nauhoittaa Chrome kehittäjätyökalujen nauhoittajalla, testejä voidaan suorittaa paikallisesti ja yrityksen Jenkins-palvelimella ja luodaan erilaisia raportteja. Nämä raportit on integroitu Jenkinsiin ja ovat katseltavissa Jenkinsin kojetaululta. Implementointia arvioidaan yrityksen antamia vaatimuksia vasten tapaustutkimuksena ja WebdriverIO toimii suunnittelutieteen tutkimuksen toteutuma artefaktina.

Testauksen teoriaa on tutkittu ja sovellettu implementointi prosessin aikana ja tulokset jaetaan yrityksen kanssa. Tärkein löytö on se, että ohjelmisto on tehty ennen kaikkea testattavuus mielessä. Suositus annetaan graafisen käyttöliittymän testauksen aloittamisesta yksinkertaisilla savutesteillä ja tulevaisuudessa tasapainottaa testejä monimutkaisuuden ja huollettavuuden välillä pitäen samalla mielessä, että automaatio ei ole kannattava ratkaisu kaikkeen.

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

School of Engineering Science

Software Engineering

Mikko Nummila

Implementing GUI testing using WebdriverIO to company's Vue.js based web application

Master's thesis

2022

83 pages, 3 figures, 7 tables and 4 appendices

Examiners: Associate professor Jussi Kasurinen and M.Sc. (Tech.) Ville Nenonen

Keywords: Design science, GUI testing, testing automation, WebdriverIO, implementation, software testing

This thesis describes the implementation of WebdriverIO browser automation test framework to the company's Vue.js based web application and development pipeline. For this implementation, a mock-up of the application backend is made using Mock Service Worker. The results are that test scripts can be recorded using Chrome developer tools recorder, tests can be run locally and in the company's Jenkins server and different reports are generated. These reports are integrated to Jenkins and viewable from the Jenkins dashboard. The implementation is evaluated against company's requirements as a case study and WebdriverIO being the instantiation artifact of design science research.

Testing theory is researched and applied during the implementation process and findings are shared with the company. Most important discovery being that the software is made with testability in mind in the first place. Recommendation is given about starting Graphical User Interface (GUI) testing from simple smoke tests and then balance between the complexity and maintainability of the tests in the future, while keeping in mind that automation is not a viable solution for everything.

ACKNOWLEDGEMENTS

I would like to thank the company for providing the topic and place to do this thesis as this was a great opportunity to do this thesis for a third party outside the university.

I would also like to thank Jussi Kasurinen and Ville Nenonen for supervising this thesis and everyone who gave comments, feedback and helped in any way.

ABBREVIATIONS

API	Application Programming Interface
AR	Action Research
BS	Behavioral Science
CI/CD	Continuous Integration and Continuous Delivery
DS	Design Science
FEDS	Framework for Evaluation in Design Science
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
IS	Information System
JSON	JavaScript Object Notation
OS	Operating System
RD	Routine Design
REPL	Read Evaluate Print Loop
UI	User Interface
URL	Uniform Resource Locator

Table of contents

Tiivistelmä

Abstract

Acknowledgements

Abbreviations

1 Introduction 8
1.1 Scope and limitations 9
1.2 Goal of the thesis 10
1.3 Structure of the thesis 10
2 Software testing 12
2.1 General testing concepts 12
2.2 Testing automation 14
2.3 GUI testing 17
2.4 Tools 20
2.5 Methods, techniques, and strategies 22
3 Research method 29
3.1 Design science 29
3.1.1 Behavioral science vs Design science 30
3.1.2 Action research vs Design science 31
3.1.3 Routine design vs Design science 31
3.2 Justifying design science 32
3.3 Design science research process 33
3.4 Research artifact 36
3.5 Evaluation 37
4 Implementation 43
4.1 Choosing testing tool 43
4.1.1 Gathering of the tools 43
4.1.2 Limiting selection 44
4.1.3 Testing and notes about the final candidates 46
4.1.4 Final choice 50

4.2 Choosing testing methodologies, strategies, and good habits 51
4.3 Implementing a solution 53
4.4 Ready solution 60
5 Results 62
6 Discussion 66
7 Conclusion 68
References 70

Appendices

Appendix 1. Testing tools

Appendix 2. Used commands

Appendix 3. Jenkins file

Appendix 4. WebdriverIO configuration file changes

1 Introduction

The longer a company builds software in its own way and tries to stay the same, if possible, the more problems, inconveniences and targets for improvement start to show up eventually. Everything develops as time goes on and especially in the field of software development that change can be quite fast. In the target company it was already noted multiple times that software testing has much room for improvement. Be it unit testing, integration testing, regression testing, GUI testing or performance testing. Currently the testing is nearly nonexistent. Only some unit tests are made in the regression testing sense to ensure that new changes do not massively break anything old. These tests are automated but for the rest of it, the testing does not exist or is done by hand in the most critical parts of the software. Especially in the custom-tailored parts of the software, there is nothing automated and only some testing is done by hand manually. Due to the nature of the company's software being very much tailored for each of their customers, it is acknowledged that not everything can be easily automated but there is still huge potential for testing automation. This automation would be in the centric parts of the software that stay nearly identical, regardless of the customer specific changes to the software. From there on the automation could continue differently for each tailored piece of software, depending on how it would fit to the company's workflow and other systems, but this is the start. The main point is that the testing that is done by hand should be minimized as much as possible in favor of testing automation and get the testing to be a centric part of the development cycle.

The point of this thesis is to focus to the GUI part of the testing process. Gather tools and information about GUI testing, select the one tool which to use for GUI testing and implement it to the software product, development environment and continuous integration continuous development (CI/CD) pipeline. But why focus on the GUI part of the testing instead of any other part of the testing process? The target company has decided to start changing its software's web-based user interface (UI) from older Java Wicket based solution to a newer Vue.js based solution and decided that now would be a good chance to make improvements to the GUI testing. As no major changes are made yet, aside of one tool that was created as a test by the company to test the new technology, now is the optimal time to thoroughly research a new solution for the company's testing needs to improve the future situation. This way the new testing solution is ready to go when the actual major changes to the GUI technology in the software product start to go through. Also, the company can take into consideration the possible needs or quirks of the testing tool to make testing easier to implement and execute, when writing the new code from Java to Vue.js. While many modern tools have quite similar structure and features, there are always things that would be good to take into consideration at an early stage when making the implementation, especially to a larger project.

1.1 Scope and limitations

The scope of this thesis is for the testing process of the target company and specifically in the GUI testing process. Many of the things that affect to the GUI testing can also be used in other testing processes, but this thesis focuses only to GUI testing process to keep it from becoming a general testing topic with thinly spread content. We investigate GUI testing tools that are easily available and reasonably often updated or maintained, so that it could be adapted into the company’s existing workflow with the certainty that the tool will still exist and be usable multiple years later. Initial gathering of testing tools will list everything that is found even somewhat relevant, but everything that has not been updated within two years will be ruled out in the first closer look up, as for a company to start using a tool, it must be updated regularly and have stable future, or it would have to be changed quite soon for another one. There are no definitive requirements for the initial search for tools and after the first filtering that rules out all outdated tools, a closer look is taken at the tool list with the requirements from the company.

First, testing tools are investigated, until one is found that would be implemented for the company. Then testing methodologies are investigated for what would improve the usage of the selected testing tool and therefore the results of GUI testing overall in the company. Here gathering is started for different testing methodologies from a wide range, compare them against each other, research previous findings about the topic and then decide which one or few would be the best for this specific scenario.

A closer look is also taken to the selected tool to see if there are any kind of changes or improvements to the company’s software code itself, that would make the usage of the testing tool easier or lower the possible barrier to entry. This is a great opportunity to do this as this thesis takes place prior to any major changes to the software and the implementation for the testing tool can be done before the changing of the UI code begins. The testing should not be seen as a chore that keeps getting pushed back, because it is too difficult or inconvenient to do.

After everything else, an implementation is done to the existing company workflow which integrates it to the development pipeline and any other systems that it can integrate and where company wants it. There will not be any report about the longevity of this project or any parts of it and this thesis ends to the implementation of the new testing process.

1.2 Goal of the thesis

The goal for this thesis is to implement a successful solution for the GUI testing process to the target company, based on research results. This requires the gathering of testing tools for a comparison and after filtering the list of tools based on the company requirements and opinions, few promising candidates are looked at closer and tested. From there together with the company a selection is made for the tool that fits best to its needs.

Then testing methodologies are researched in the context of GUI testing, to see how testing should be done for the GUI more effectively in the first place and what methods would go well with the selected tool. Few of the top methods will be suggested to the company. To test their GUI more effectively generally and how to use the selected tool to achieve the wanted results and how to overall use the tool according to these methodologies.

Finally, an implementation of the testing tool is made to the company's development workflow. It will be integrated to the CI/CD pipeline and development environment. An example use case is made by using the implemented tool and selected testing methodologies, to make tests to the one software application that is already made with Vue.js to see that everything works properly without any problems.

1.3 Structure of the thesis

Thesis starts with an introduction. This is followed by theory about GUI testing, testing automation, testing tools and general testing concepts in the second chapter. Third chapter explains the research method of this thesis and covers why it was chosen. Fourth chapter consists of the implementation of the practical part in this thesis. It explains the selection process of the testing tool, methods for GUI testing based on the finding in the second chapter, actual implementation process to the company product and results of the practical part. Fifth chapter presents the results of the design science part of the thesis and the evaluation of the artifact. This is followed by sixth chapter that discusses about related works, how this work relates to design science and possible risks. Thesis ends at the seventh chapter which contains the conclusion of the thesis.

2 Software testing

Software testing is an important part of the development process if one wants to release quality software. This can also straight up translate to the development costs as the earlier an error is found, the cheaper it is to fix it (Myers et al., 2004). This is especially true as the testing can be as much as 50%-75% of the total cost of the product release (Blackburn et al., 2004). It even follows to the GUI, since it can be 60% of the app code (A. M. Memon, 2002). Having such high costs, many try to automate the testing process to save resources, but if it is not done right and in right places, it only causes additional problems.

In this chapter we go through relevant theory about testing in the context of this thesis. It means that everything is looked at from the GUI point of view. After theory about general testing, more is talked about testing automation, followed by a section about GUI testing specifically. Then go through tools for testing and at the end look about different testing methods.

2.1 General testing concepts

Testing is a large part of software engineering and because of that it is a continuously researched subject. This has also helped to form standards for software testing, that define everything testing related and there are multiple parts that are focused on different aspects of testing like general concepts, testing processes and test techniques (IEEE, 2022, 2021a, 2021b). These standards are not meant to be followed word for word as it would be near impossible and just a waste of resources, but they can however act as a starting point and work as a guideline to construct your own testing process.

Testing can be defined multiple ways, depending on who you ask. According to Pettichord any testing can be described in five dimensions that are testers, coverage, potential problems, activities and evaluation (Pettichord et al., 2011). Myers defines it as “Testing is the process of executing a program with the intent of finding errors (Myers et al., 2004)”. Software testing standard defines test as “activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component (IEEE, 2022)”. Testing itself is a destructive process as a test case is a successful one if it finds an error (Myers et al., 2004). Therefore a good tester must be able to think differently compared to the programmer who coded the software, but still be a good communicator as it is the key to everything (Pettichord et al., 2011). Myers presents ten vital testing principles that should be taken into consideration, these can be found in Table 1. While some of these principles might be obvious, they are easily

Table 1. Vital testing principles (Myers et al., 2004)

Number	Principle
1	A necessary part of a test case is a definition of the expected output or result.
2	A programmer should avoid attempting to test his or her own program.
3	A programming organization should not test its own programs.
4	Thoroughly inspect the results of each test.
5	Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6	Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7	Avoid throwaway test cases unless the program is truly a throwaway program.
8	Do not plan a testing effort under the tacit assumption that no errors will be found.
9	The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10	Testing is an extremely creative and intellectually challenging task.

underestimated and overlooked. (Myers et al., 2004) When testing process takes into consideration all these principles and even follows most of them through, becomes testing easier, and more thorough.

As to why users find bugs in the software, there are many reasons why these errors were not found before like user executed untested code, order of execution is different from testing, user applied untested combination of input values or the users operating environment was not tested (J. A. Whittaker, 2000). Important thing to mention is that testing a program completely is impossible, as the number of possible test cases becomes easily near infinite. It is also impossible to test a program to show that errors are not present as even working programs can contain errors. (Myers et al., 2004) Therefore it is important that testing is planned properly, and tests are designed to find the most errors possible. The question to be asked is that:” What subset of all possible test cases has the highest probability of detecting the most errors (Myers et al., 2004)”. Whittaker proposes four phase approach to testing as follows; model software environment, select test scenarios, run and evaluate, and measure progress (J. A. Whittaker, 2000). Nowadays there is the standard for test processes that can be referenced to start a testing process (IEEE, 2021b).

Test plan should be created before any testing occurs to describe all testing related operation. This way there is a written document where you can fall back to, in case something unexpected happens or something is unclear. According to the definition for test plan in (IEEE, 2022), test plan includes test objectives, means to achieve them, schedule and organized activities for test items. Myers says that components of a good test plan are objectives, completion criteria, schedules, responsibilities, test case libraries and standards, tools, computer time, hardware configurations, integrations, tracking and debugging procedures and regression testing (Myers et al., 2004). Test plan can also be defined as a set of ideas that guide the test process and then test plan document is a document intended to convey those test plan ideas (Pettichord et al., 2011). The basic gist of the test plan is that there should be one, but it only needs to be as complex and comprehensive as it needs to be. There is no need to make it include all possible information that it could hold, as it would be just a waste of resources.

2.2 Testing automation

Testing automation is on the rise in the industry, and everyone seems to want to take part in it. Studies about the industry show that testing automation has increased significantly, reason being that there are now better tools available and organizations change towards agile practices (Hynninen et al., 2018). However the adaptation of automation seems to be rather demanding and the lack of resources provided to testing may be the reason that the average adaptation falls lower than what can be seen from literature (Kasurinen et al., 2010). Companies have also published their own research like one company shares their common pitfalls in automated regression testing, and notice a strong relation between informal risk management and successful automation project (C. Persson and N. Yilmazturk, 2004). Many companies, while wanting to get into the automation, keep thinking what they can automate instead of the more important question what they should automate. Study from Karhu found that automation is very subjective and varies wildly between different cases from company to company (K. Karhu et al., 2009). Testing can be divided to automatic and manual testing as there are things that are suitable for each of them (Kasurinen et al., 2010). However automatic testing cannot replace manual testing as they are different in nature (S. Berner et al., 2005). Experienced testers recommend semi-automatic testing where core tests are automatic and other tests are manual (Mahmud et al., 2014).

According to Zhyhulin the goals of the testing automation are improvement in efficiency, as for example the tests can be run after hours, reduced testing time, transparency of the testing results and reduced testing costs. Transparency would come from the fact that everyone has access to the testing reports. (Zhyhulin et al., 2022) The biggest factor for driving automation would be costs, as if there were no possibility for savings through automation, no one would try to do it. Most companies make decisions ultimately based on money and this is no different to that end. Karhu says that major benefits from testing automation are improved quality through better coverage, but this requires human involvement for selecting test cases. More testing can be done at the same time, which also saves money. (K. Karhu et al., 2009) Tests would be repeated more often, which means more usage, so they would be more cost effective (S. Berner et al., 2005). Other reasons to automate testing include contractual or regulatory reasons, tests will be delivered to customer, product requires backwards compatibility and some products can only be tested through automation (Pettichord et al., 2011).

Then for the advantages that the automation can bring, there are at least as many disadvantages that need to be considered for your situation. Possibly the biggest disadvantage of automation is the overall additional costs that it brings, as those are mentioned as disadvantages in nearly all papers. The test ware maintenance is hard, as architecture is often undocumented, there are missed opportunities for reuse of code and test ware can be poorly structured and untested (S. Berner et al., 2005). Automation causes additional training, maintenance, implementation and unreliability if implemented poorly, which all increase costs (K. Karhu et al., 2009). Projects for automation require programming, testing and project management skills (Pettichord et al., 2011). Kasurinen argues that automation is more of a quality control approach rather than frontline testing approach, although that can also be an advantage depending on your situation. Maintenance and development costs are the biggest obstacle regardless of the size of the automation project, and automation is possible to implement but requires a lot of effort. (Kasurinen et al., 2010) In the end all the mentioned disadvantages can be brought back to the one single problem that is increased costs, but if everything is done properly the advantages of automation can outweigh the disadvantages.

Basically, there is no such thing that could not be automated, but it could just create more problems, be more expensive and be more difficult to handle compared to keeping it manual. There are clear areas that could or should be automated and then there are clear areas what just should not be automated at all. According to Zhyhulin you should never automate unstable software that is still under development, rarely used scripts and any analysis and documentation of the program code. These will only cause problems in the future. Suitable targets for automation are hard to reach places in the system, frequently used functions, routine operations, long end to end scripts and data validation that requires accuracy. (Zhyhulin et al., 2022) Good candidates for automation are smoke tests, which broadly verify functionality, and unit tests, and what usually require automation are load-, performance-, configuration-, endurance-, combination error testing and race conditions (Pettichord et al., 2011). From Karhu, good for automation are generic and independent products that are similar with each other in tech, high reusability, and low human involvement. Bad for automation is the opposite, plus the rapid development of the underlying tech, which can make automation systems obsolete. (K. Karhu et al., 2009) Some of these points are also shared by other companies such as automating unstable objects (C. Persson and N. Yilmazturk, 2004). GUI is the hardest part to automate and should be avoided, but there are exceptions (Pettichord et al., 2011).

Part of the automation is the program to be automated itself and there are different approaches, good things and possible problems when applying automation. But in order make sure that testing automation can be implemented easily, the testable program must be made testable. Testability should be taken into consideration in the design phase and code should be testable (Blackburn et al., 2004; J. A. Whittaker, 2000; Pettichord et al., 2011; S. Berner et al., 2005). It is also a huge concern in the industry (Kasurinen et al., 2010). Testability is visibility and control, which means features that help testability like access to source code, logging, diagnostics, error simulation, test points, event triggers, reading obsolete data formats, test interfaces, custom control support and permitting multiple instances (Pettichord et al., 2011).

Before automation, tests should be already designed, as otherwise there could be much wasted work, if tests change in the middle of automation (Pettichord et al., 2011). In the industry, testing has shifted from the design phase to later phases like acceptance testing, because features are added later so testing time is shorter (Hynninen et al., 2018). To make the testing automation easier it should be started as early as possible and be built alongside the software (Pettichord et al., 2011). Berners advice is that when starting automation, a proper strategy should be chosen. This dictates goals, automation approach and evaluation. Also, good engineering practices should be used. When testing is possible, tests should be run to avoid their deprecation. (S. Berner et al., 2005) Pettichord adds that things that affect to the strategy are testing requirements, product architecture and staff skills (Pettichord et al., 2011).

If automation is implemented poorly there are many possible problems that can occur. These include poor coverage, false alarms, false test results, tests may no longer be interesting and tests might not do what you thought (Pettichord et al., 2011). Pettichord continues: ”capture replay test automation is an ice-covered slope with brick wall at the bottom”, meaning that it is a great addition to your testing but very bad as the only solution. There are also possible concerns from the design of the automated tests, like have they been set up correctly, are expected results specified, do they notice potential errors or side effects, do they recover from failure and can they interfere with each other (Pettichord et al., 2011).

2.3 GUI testing

GUI is an extremely important part of the software product as it is almost always the only part that the customer sees. If the GUI is broken, the customer can easily lose trust to the software (Moreira and Paiva, 2014). If it is a web page, users can be turned away by problems or inconveniences on the site (Myers et al., 2012). Many applications have changed from a desktop application to a web application and that means the GUI has changed as well. Not necessarily too much from the outside as they can still be quite similar, but the underlying technology is completely different. These differences make the testing of these two types of GUIs a very different challenge.

The main characteristics of a GUI are graphical orientation, event-driven software, extremely large input space, hierarchical structure, variety of graphical objects and their attributes, object-oriented programming and a wide range for user interaction with the GUI (Kandil et al., 2012). These characteristics make the testing of GUIs quite difficult because there are so many things that need to be taken into consideration. That makes it much harder to test than normal command-line interface testing (Singhera et al., 2008).

Goals for GUI testing process can include a variety of things. According to Memon possible goals are task automation, efficient overall test cycle, robust testing algorithms and portability. Robustness meaning that the tests can recover from unexpected states and portability meaning that the tests can be easily run on other alternative platforms. Finally, the process should be general enough that it can fit to a wide range of GUIs. (A. M. Memon, 2002) Singhera shares some recommendations and best practices for GUI testing. Each GUI element should be uniquely named, and they should be well defined, testing should be planned around the testable object carefully, the testing activity should be performed in layers, tests should be organized hierarchically similarly to the object under test, a tool capable of addressing elements by symbolic name should be used to create tests, and data for verification should be captured accordingly from only affected attributes. Additionally, scripts should be as short as possible, test just one thing, be clearly described and consist of three clear sections: preparing, executing, and cleaning. They should also be callable from other scripts to improve reusability and have separate scripts for emulating user action and for data verification for easier porting of scripts to other platform. (Singhera et al., 2008) A paper describing Selenium also points out that it is a good idea to add unique identifiers to elements and refer to those identifiers (Bruns et al., 2009). Some of these recommendations are clearly more focused on desktop applications but are still relevant and a good base for building a good more testable GUI.

Differences between desktop application GUI and web application GUI are many and all the same testing methods won’t work on both GUIs. Kandil opens the differences between them. Web application can have dynamic pages that require interaction in addition to traditional ones. They are installed across network and can have different control flows. User can navigate to any page by writing its path to browser. There is not necessarily one specific GUI for the user as there are different browsers and they can affect things differently. Because of these reasons the traditional testing methods won’t work on web GUIs. (Kandil et al., 2012) These thoughts are also shared by Di Lucca, in the heterogeneous execution environments and components of the software. Additionally there can be large number of users distributed around the world and the software may generate components at runtime. (Di Lucca and Fasolino, 2006) Web also introduces wait times, because the application could work but not properly unless it waits something else to complete before it (Almenar et al., 2016). Myers shares a list of challenges about testing internet applications. Large and varied user base, which includes differences in skill level, operating system, browser, device, and connection speed. In business environment all possible calculations involving money and their transactions themselves, tracking profiles and third-party connections. Other general challenges are all possible locales like language, currency and time, security against attacks, denial of service and phishing and testing environments and their duplication. (Myers et al., 2012) All of these affects to the GUI testing, some straight and some indirectly. In conclusion all previous methods to test GUI must be adapted to the more dynamic nature of the web (Di Lucca and Fasolino, 2006). It is also harder to automate the dynamic user interactable interfaces (Simon Bisson, 2020).

When testing GUIs there are some common pitfalls that should be avoided. Traditional coverage criteria such as event coverage is not good for GUIs as it is impractical to test all possible events, assertion should be made properly so that intermediate values are also correct in addition to the final value and changes to the GUI can break the tests completely if the tests are done poorly (A. M. Memon, 2002). Testing should also take into account how the software interacts with the browser (Di Lucca and Fasolino, 2006), and make sure that it tests the application and not the browser (Almenar et al., 2016).

Testing of GUIs can be categorized to two parts, functionality testing and usability testing. Myers opens this as functionality is tested with testing tools similarly to any other testing, while usability is tested with human testers completing given tasks and then taking feedback from the testers through interviews. This gives the developers hints about possible design flaws and software ergonomics mistakes. Important part is the detailed and accurate gathering of data and its analysis. (Myers et al., 2012) Process for implementing functional testing should include the determining of the coverage criteria of what to test, generate expected inputs and outputs, execute tests, verify outputs and determine if GUI was tested adequately (A. M. Memon, 2002).

A good test suite should have simulated user generated events, layout changes should not affect the system and test coverage criteria should be present to ensure all code is tested (Kandil et al., 2012). As for test cases there are number of studies that have presented their approach to test generation, being either a new tool, methodology or framework. Many of these are designed for desktop applications in mind, focus to a tool that is built for the purpose of that study and has not seen much development after that. Zhu presents a tool that integrates test generation to other testing functionality and focuses on desktop GUI (X. Zhu et al., 2008). Generation of tests based on ripping the application GUI has several studies such as (Molnar, 2012), (Memon et al., 2005) and (Carino and Andrews, 2015) but they are designed for desktop applications that have no dynamic elements. GUI ripping is creating a path tree of possible event flows in the GUI. Before the method of ripping GUI, a technique was presented where tests were generated from given start and end points (Memon et al., 2001). Modeling based test generation has also being studied like generating tests from low level prototypes (Hellmann et al., 2010) and from proper unified modeling language models (Vieira et al., 2006). These approaches are best utilized when the software is in its design phase or early development. One approach uses built in accessibility features to manipulate GUI in a uniform way and generate tests from desktop applications (Conroy et al., 2007). For a more modern look (Ermuth and Pradel, 2016) presents test generation from traces that are recorded with a browser addon and (Mattiello and Endo, 2021) presents a tool which includes model based test generation that uses page object models for readability. Nowadays many tools can generate the code simultaneously while recording and many modern tools that focus on web have the possibility to use page object models.

There can be seen a lack of UI testing on web and mobile platforms. Reasons being a lack of documentation and proper tools, significant maintenance effort and limited perception of benefits (Alégroth et al., 2021). To combat this, resources for testing should be increased and be taken into consideration overall. Also, knowledge about the benefits of testing should be brought to a wider audience.

2.4 Tools

Testing tools have been around for a long time and nowadays there are literally hundreds of them. A study by Hynninen shows that the popularity of testing tools is rising in the industry when compared to previous years. Overall usage of the tools is increased and the most popular categories of tools are unit testing, defect reporting and test automation. (Hynninen et al., 2018) A mapping study about GUI testing field from 1991-2011 shows that the most used tool was GUITAR using Java language (Banerjee et al., 2013). Nowadays this tool is basically abandoned but for a time it was very popular. Many studies mentioned in previous section used GUITAR in them for ripping the GUI. Many tools were also created in research purposes like CoTester (Mahmud et al., 2014) and TESTAR (Almenar et al., 2016). A survey to the Selenium ecosystem shows that the most used Selenium project is WebDriver used with Java code driving chrome browser, subject under test was web application in enterprise setting testing functionality, locator usage was based on id, pipeline included Jenkins and other parts were testing project specific (García et al., 2020). There are also many studies that compare testing tools either generally like (Sabev and Grigorova, 2017) in a larger scale and (Okezie et al., 2019) in a smaller scale, or focuses more towards a specific goal like regression testing (Zhyhulin et al., 2022). Whatever the specific goal may be, they all came to the same conclusion in one thing that there is no perfect tool. Every single tool has its own advantages and disadvantages what should be considered when selecting a tool for a project.

When selecting a testing tool, it should be selected based on compatibility, familiarity and service (Pettichord et al., 2011). Other recommendations are keeping the platform, size and budget of the project in mind when selecting (Okezie et al., 2019). Many websites list tens or sometimes even hundreds of different tools for a specific category. These lists are a good starting point for searching a testing tool. It can also be possible that just one tool is not enough for the project and nowadays many tools are designed to work together with other tools so that anyone can build the right setup for their project. For any larger project it is also important to find tools that have good history and ensure that the tools are mature enough to be usable from stability, maintainability, and feature standpoint, but also make sure that these tools have bright enough future so that they will be supported and won’t be abandoned too soon after being adapted to a project. A huge part of the tool is also proper and comprehensive documentation.

Modern tools come with varying capabilities and most of them are highly integrable. Some are created to be platforms where a solution from other tools can be built on, some are created for one thing only such as reporting, some tools are made to be a complete package, some have their own GUI and some work only in the command line. Many modern GUI testing tools are focused for testing websites and cannot even run on a desktop application GUI. Most of these tool’s work by communicating with a browser through a specific browser driver that is maintained by the developer of the browser. The driver uses the browser from outside like any user would. This way the tool can focus on its own thing and same tests can be used with different browsers. Each tool has their own syntax and selector for creating tests and navigating the website. There are also tools that have their own language to differentiate from other tools. These tools try to make it easier to write tests with their own language, but often it is more painful to learn new language than use known one with possible small hinderances. It is also recommended to avoid proprietary languages (Pettichord et al., 2011). Many GUI tools also support page object models, that can be used in multiple tests after their creation like components in GUI code. This helps the reusability of code.

For GUI testing several tools offer the option to use record playback functionality. This means that user can record the test by doing it manually first and then play it again later. With this it is easy to create tests but it has some downsides as also mentioned by (Pettichord et al., 2011). The code generated by the recording functionality is quite specific to make sure that it follows users' actions precisely. This means that the tests are easily broken by the smallest change to the GUI and need to be rerecorded (Blackburn et al., 2004). Absolute paths are often used to identify elements in recordings so any change that affects the path can break the test. Therefore, recorded tests should be edited to be more adaptable to changes. Recordings are great for creating tests fast and testing the tests effect before making it more permanent and some simple commands like clicks and inputted data don't necessarily even need any changes for the test to be good. It is recommended to have a small pilot project where the tool can be tested properly before adapting it to a larger part of the software (Blackburn et al., 2004; Pettichord et al., 2011).

2.5 Methods, techniques, and strategies

Testing techniques can be categorized in multiple ways. First division is between human testing techniques and computer testing techniques. From there computer testing techniques can be divided to black-box, white-box, and grey-box techniques and from there to individual testing techniques.

Myers opens this as human testing techniques are code inspection, group walkthrough, desk checking and peer review. Code inspections and walkthroughs are quite similar that a group of people go through code and try to find and fix errors, but inspections contain a checklist. Desk checking is like a one-person code inspection that can be done to a colleague and peer review evaluates code and allows programmers to self-assess their skill. The advantages of human testing techniques are that they frequently expose a batch of errors, are good for certain types of errors and most importantly people other than the author are involved in the process, but they are not good at finding high level design errors. Also just reading code aloud can be an effective technique. (Myers et al., 2004)

Then there are black-box, white-box, and grey-box testing techniques or strategies as they are sometimes called. These mean how the tester sees the software under test. Black-box meaning that the software is completely unknown to the tester and just inputs and outputs are seen, while white-box means that tester can see everything in the software inputs, outputs, and internal workings (IEEE, 2021a). Grey-box techniques fall in between the others and are everything that is not completely either black-box or white-box technique and if method of testing is used which combines black-box and white-box techniques (IEEE, 2022). According to Myers black-box testing is data-driven and input/output-driven testing as we don’t know the internals of the program and finding all errors requires exhaustive input testing. White-box testing is logic-driven testing as we do know the internal of the program and finding all errors requires exhaustive path testing. In both cases, there are near infinite amount of possible inputs and paths so complete testing is impossible. (Myers et al., 2004)

Individual techniques themselves are logic coverage testing, equivalence partitioning, boundary-value analysis, cause-effect graphing and error guessing (Myers et al., 2004). Pettichord divides techniques as people-based, coverage-based, problem-based, activity-based and evaluation-based techniques, these can be seen in Table 2 (Pettichord et al., 2011). These

Table 2. Testing techniques (Pettichord et al., 2011)

technique	activity
people-based (focus on who does the testing)	user testing, alpha testing, beta testing, bug bashes, subject-matter expert testing, paired testing, eat your own dogfood
coverage-based (focus on what gets tested)	function, feature/function integration, menu tour, domain testing, equivalence class analysis, boundary testing, best representative testing, input field test catalogs, logic testing, state-based testing, path testing, statement and branch coverage, configuration coverage, specification-based testing, requirements-based testing, combination testing
problem-based (focus on why you test)	input-, output-, computation-, storage/data constraints
activity-based (focus on how you test)	regression-, scripted-, smoke-, exploratory-, guerilla-, scenario-, installation-, load-, endurance-, performance testing

evaluation-based (focus on how to tell whether test passed or failed)	self-verifying data, comparison with saved results, comparison with specification, heuristic consistency, oracle-based testing

categories have some overlapping as Myers describes techniques themselves and Pettichord describes partly actions done in these techniques. For example, logic coverage testing, and coverage-based techniques mean basically the same things but activity-based techniques describe the different testing activities such regression testing which can be done as either black-box or white-box testing depending on whether the internals of the software are known.

Coverage-based testing has several forms. There is statement-, decision-, condition and multiple condition coverage. Myers explains that statement coverage means that each statement in the program is executed at least once, but this is near useless as it is easily fulfilled and rather weak criteria. Decision coverage that is also known as branch coverage means that every decision is executed at least once and can usually also fulfill the statement coverage as every statement is part of a decision but there are exceptions. If program has no decisions, has multiple entry points, or has code in error handling blocks, statement coverage may not be fulfilled. Coverage that is sometimes stronger than decision coverage is condition coverage which means that each condition to a decision gets all possible outcomes at least once. This is only sometimes stronger because it does not necessarily execute each statement and go through multiple entry points. Solution is decision/condition coverage that combines them, meaning that each condition in a decision and each decision takes all possible outcomes at least once and each entry point is invoked at least once. Weakness of this is that some conditions can mask other conditions and prevent them from executing. Final coverage is multiple condition coverage meaning that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. (Myers et al., 2004)

Equivalence partitioning and boundary value analysis is often used together. Myers explains that equivalence partitioning divides input domain to equivalence classes that have different inputs so that only one unique testcase needs to be done from each equivalence class. Boundary value analysis adds to that by using inputs in testcases that are at the edge of the input space because boundary conditions have a higher payoff compared to other places. This has the weakness that it takes into consideration only one input and does not care about combinations of inputs and erroneous inputs can mask other inputs. Error guessing can partly be mentioned here as it is not a set technique itself and uses parts of other techniques. It is mainly just guessing and relies heavily to the expertise and experience of the tester. Some situations are just more error prone than others like calculations involving zero and experienced tester knows that. (Myers et al., 2004)

The weakness of combined equivalence partitioning and boundary value analysis can be solved with cause effect graphing. Myers gives six step systematic way to handle cause effect graphing. First specifications are divided to workable pieces. Second step is identification of causes and effects. Third, semantic content is analyzed and transformed into Boolean graph linking causes and effects. Fourth, graph is annotated with constraints from impossible combinations. Fifth, methodically tracing state conditions, converting graph to a decision table, where a column is a test case. Sixth, test cases are created from columns. This process can be quite difficult depending on the software, especially the graph to table conversion, but it can reveal incompleteness in specifications. (Myers et al., 2004)

These different testing techniques are the base of many more advanced and specific techniques that focus to a specific part of the software. For GUI testing specifically there are several model-based techniques that can be used for testing and test generation. Memon presents different test generation techniques on GUI applications with state machines, workflows, pre- and postcondition models, event sequence-based models, probabilistic models, combinatorial models and hierarchical models (Memon and Nguyen, 2010, p.). These techniques are not meant for web applications, but some ideas can be taken from them. Silistre reviews different models used in model-based GUI testing and proposes converting them to event sequence graphs (Silistre et al., 2020). Memon presents an event flow model for GUI testing (Memon, 2007). Moreira presents pattern-based GUI testing approach that uses generic strategies to test UI patterns (Moreira et al., 2017). There are also requirements for model-based testing from test, component, system, and process engineer presented that help to set it up (Santos-Neto et al., 2007). Problem with these model-based techniques is that they require the GUI to be modeled or ripped to create the models to move forward and that does not work well with dynamic elements in the GUI. For that reason, they are less useful in web-based GUI applications and especially modern ones that can have a lot of dynamic content in them.

For usage of any technique or combination of them there should be testing strategy that is followed. Case study from the industry shows that most follow some systematic process or method (Kasurinen et al., 2010). According to Pettichord a strategy contains the reasoning for what is done and a good one is product specific, risk-focused, diversified, and practical. It should evolve with time as testing early is easier and simple tests are enough while later it becomes harder to find errors as the program is more resilient. Commitment to a single strategy early on should be avoided. Testing can also be thought as levels to simplify discussion. (Pettichord et al., 2011) Myers tells the following strategy that is based on his classification of techniques. First start with cause effect graphing if the program has combinations of input conditions, use boundary value analysis in all cases and supplement these with equivalence partitioning. Use error guessing to add more cases, examine the program logic regarding tests and add tests if coverage criterion is not met. This is a reasonable compromise, since it is not perfect. (Myers et al., 2004)

To decide when testing is done, a completion criterion is used. Because otherwise testing would be never ending, having one is necessary. According to Myers two most common completion criteria are to stop when scheduled time ends and to stop when tests no longer find errors. There are more criteria like base the completion to a specific test-case-design method, state completion in a positive terms like number of errors found and plot the number of errors per unit of time. Possible problem with these is that what is the number of errors as it can be easily overestimated. The best theoretical criterion is to combine all the above criteria but it depends on the software project what is really feasible. (Myers et al., 2004) Certain code coverage from testing can also be a completion criterion. From the mapping study it can be seen that the most used evaluation method was feasibility of a technique and evaluation metric was number of faults detected (Banerjee et al., 2013). When considering web-based applications possible criteria are also page coverage and link coverage (Di Lucca and Fasolino, 2006). No traditional coverage criteria fits easily to GUI testing so these must be adapted case by case basis (Kandil et al., 2012).

There are different approaches for testing web applications and then there are different approaches for testing GUI. When these two are combined an approach needs to be adapted to accommodate elements of both. For web applications some possible testing approaches are systematic testcase generation using event-based model, models that produce test paths for good coverage to reduce time and model web applications to finite state machines (Kandil et al., 2012). Web applications can also be seen as three main layers that can be tested individually and these are presentation, business and data layers (Myers et al., 2012). For some

GUI testing approaches there are reverse engineering techniques to automatically generate testing model, genetic algorithm to test GUI functionality and generate tests from graph models created by approximating possible sequences and generate GUI tests in batches (Kandil et al., 2012). Problem in these is that they are mostly meant for static desktop applications and the main keyword here being static. Conclusions from Kandil about the previous techniques are that there is no single technique that can cover all the bugs and test-driven development cannot be directly utilized without defining appropriate methods. Genetic algorithms might be successful in creating all paths, but enterprise sized applications have huge number of tests and large amount of them have no importance. Finite state machine is best for testing coverage and input validation, event flow and system requirements approaches were good for test first design by effectiveness. (Kandil et al., 2012)

Myers proposes strategies for testing of e-commerce web applications layers and encourages to test each layer separately as each layer has their own characteristics that encourage this. However, each layer has the following that should be tested: usability and human factors, performance, business rules, transaction accuracy, data validity and integrity, system reliability and network architecture. (Myers et al., 2012) While all these points like transaction accuracy may not apply to all web applications, everything else should be noted and at least be aware of them when testing web applications. Myers continues about presentation layer testing that it means finding errors in the front end, meaning GUI, it can be very labor intensive and can be further divided to three categories. First is content testing that focuses on overall aesthetics, fonts, colors, spelling, content accuracy and default values. Second is website architecture and it focuses on broken links and graphics. And third is the user environment that focuses on web browser versions and operating system configuration. Since browser compatibility can be the most challenging aspect of web application testing. (Myers et al., 2012)

Kandil presents a new approach taking into consideration the flaws of other methods that are already mentioned. In this approach a grey-box technique is used to generate three sets of tests from system requirements. One set is directed to functionalities of GUI independent of business logic, one set is for application logic without the GUI and one set is for screen transitions. This would guarantee complete test coverage with least amount of tests. (Kandil et al., 2012)

From everything that is discussed the one conclusion to make is that there is no one perfect technique and it is highly dependent on the testable software. Strategy should be decided with completion criteria early on and then start looking into what methods are fitting for this specific application taking into consideration as much as possible from the testable software. When testing modern GUI, older techniques do not work straight up and need to be adapted for the environment. One of the main points being dynamic content that cannot be straight up modelled and requires additional work. Testing should be divided as small pieces as possible to make it easier and apply the basic techniques like boundary values and equivalence classes wherever possible. Everything should be planned carefully from the start, develop tests before coding the software to find possible design errors and evolve the testing plan along the development of the software. The earlier the errors are found and fixed the more cost efficient it is, and quality of the software rises.

3 Research method

For this thesis, design science (DS) was chosen as the research method as it was seen as the best fit for this topic. It has been used in information systems (IS) research for years since it emerged to this area of science. There are a lot of research to the DS itself, such as guidelines for how to use it (Hevner et al., 2004), and how the DS research process works (Peffers et al., 2006). This thesis uses DS as a research method, with the selected testing tool as an artifact, trying to solve a problem that is the lack of testing. In this chapter we go through the thesis from the research methods point of view. Justify the use of DS as a research method by comparing it to routine design (RD), action research (AR) and behavioral science (BS). Then go through the guidelines for DS research presented by Hevner (Hevner et al., 2004), and finally take a closer look to the artifact of this thesis and its evaluation.

3.1 Design science

Design science is a rather new method of research in the field IS of research. It has been present in other fields, but really came into the IS space in the early 2000s, thanks to research that proposed this methodology. (Peffers et al., 2007) presented DS as a usable methodology in IS research, (Hevner et al., 2004) proposed guidelines for DS in IS research, and (Peffers et al., 2006) proposed a process how to use DS in IS research. The idea for this has been in the minds of others for some time. As predecessors for DS, such as systems development as a methodology presented by Nunamaker et al. (1990) or modeling design process presented by Takeda et al.(1990) are considerably older and have later developed to what DS is today. Nowadays there are much more literature about DS such as (Alan. Hevner and Chatterjee, 2010), research using DS and research about DS, (Peffers, 2012) is just one collection about the topic and it can be seen that DS is well accepted and drive to improve it is high. DS is increasingly recognized as equal to BS in the IS field (Salvatore T. March and Veda C. Storey, 2008).

When talking about design science, it must be ensured that we are indeed talking about DS as it can easily be confused to especially routine design, since they are quite similar (Alturki et al., 2012). There are also behavioral science, routine design, and action research, but they have more differences compared to DS. Hevner also points out that within DS there are two classes of research, design research and researching design (A. Hevner and Chatterjee, 2010). According to Hevner, researching design is an iterative process that focuses on studying designs, designers, and design processes with the goal of generating domain-independent understanding of design processes. It has difficulties to provide unambiguous and universally accepted design processes and is mainly allied with cognitive science and professional fields like architecture and engineering. Design research is often used in a specific application context and can be clearly influenced by the opportunities or constraints of the application domain. It may need additional research to generalize its findings, values research outcomes that focus primarily on improvement of the artifact in a specific domain and extendedly seeks to broaden the understanding. Design research is mainly allied with computer science, software engineering and organizational science. The biggest differences of these two classes of research are their allied fields and that design research emphasizes the domain where it takes place. (A. Hevner and Chatterjee, 2010)

3.1.1 Behavioral science vs Design science

The other concepts mentioned before: action research, behavioral science and routine design are central concepts around the debate about the core of IS research. And how does DS compare to these other concepts? Alturki tells that BS and DS have some similarities. They complement each other by the following. DS provides novel artifacts and BS justifies theories and explain and predict the phenomena, both aim to increase the relevance and rigor of the research and kernel theory is one complimentary form between them. (Alturki et al., 2012) They also have their differences as Alturki follows. DS research focuses on the question “what can be?”, developing design knowledge used in constructing solutions and interested in utility. Is prescription-driven, meaning that it aims to search improvement in performance, it looks forward to create possibilities with new artifacts, it is known as knowing through making and it aims “exploration and validation of generic cause-effect relations (Winter, 2008)”. (Alturki et al., 2012) DS is problem-focused, as challenges for it are building and evaluating artifacts (Salvatore T. March and Veda C. Storey, 2008). Alturki continues that, where BS focuses on “what is?”, explaining causality and interested in truth. It is description-driven, meaning that it aims to understand nature of the subject, concentrate analysis on existing system, it looks back to explain the past theories, it is known as knowing through observing and it aims “construction and evaluation of generic means-ends relations (Winter, 2008)”. (Alturki et al., 2012)

3.1.2 Action research vs Design science

Design science is much closer to action research than it is to behavioral science. There is even debate that is DS and AR the same thing. AR and DS share the similar research process cycle, content, starting point, goals, proactive approaches, iteration, interfering and changing the real world, solving a problem, contributions to knowledge and practice and rigor and valuation in results (Alturki et al., 2012). Alturki follows, even if they share so much between them, they do have their differences. It is argued that DS is a research orientation and AR is more of a method. DS research’s interest is inventing new artifacts and technical problems and innovations and users participating in DS research can be virtualized. DS tries to create an artifact to learn new information and solve a class of problems with a generalized solution or create inventions. Design science is discovery through design. (Alturki et al., 2012) Action research’s interest is to understand and change complex reality, socio-technical problems and innovations and users participating in AR always participate. AR tries to develop action to change organization to build new information and is focused on organizational changes that include collaboration between client and researcher. AR is discovery trough action. (Alturki et al., 2012)

3.1.3 Routine design vs Design science

Third one of the concepts is routine design and it is quite like design science. Alturki explains that the main thing to differentiate the two is that DS produces knowledge and RD does not. RD uses existing knowledge to create a solution to its problem. DS invents technology to a generalized, abstract or a class of problems for a class of organizations and stakeholders in the academic domain. RD is technology application and uses technology to solve a specific problem for a specific organizations and stakeholders in the professional domain. (Alturki et al., 2012)

3.2 Justifying design science

Looking at this thesis, there are things that can fit to the other research concepts and not just design science. However, these points can be argued for the side of DS, depending on how the problem and solution is approached. Considering the goal of this thesis, the steps shortly are to find a testing tool, find proper methods to use it and implement it to software. BS is more focused to the past and if there were prior research to this exact problem for this exact software, this thesis could be largely based on that and be BS oriented in nature. Or if there were a similar problem for similar software that is solved, solution implemented and it could be observed to understand the truth and knowledge behind it, this would be more in the line of BS as it focuses on something already existing. However, this problem is exactly the other way around for the use of DS. There is no prior solution for this exact problem for this exact software, so it is about creating new knowledge for this, and it falls under DS. Focus on improving something existing is also one of the main points of DS, and here we aim to improve the company's software through testing with the help of the results of this thesis.

As for action research, it is like DS as mentioned before, but the main point when considering this thesis is that AR focuses on organizational changes. This same research topic could be done from the point of AR, but it would be completely different kind of research. DS focuses on creating an artifact to solve a problem and gather knowledge. Here the artifact is the selected testing tool and the problem to be solved is the lack of- and improvement of testing. Also, another point to differentiate is that, since we search a tool and methods to implement and improve testing, this is a technical problem. AR solves socio-technical problems with collaboration between researcher and client. This thesis could be completed without collaboration, so considering all of this, it is more about DS rather than AR.

Routine design also has similar points to DS as mentioned before, but the biggest reason for this thesis to use DS is the usage of knowledge. RD does focus on specific problem like this thesis does and DS usually tries to create a generalized solution, which this thesis is not exactly, but we are creating knowledge rather than using existing one. For example, RD would be using this thesis as base to implement similar testing solution to the company's other software product. DS is about creating the base research that RD uses to implement solutions to problems. Even if there are small parts of DS that do not fully fit this thesis topic, when comparing DS and RD in this context, this falls under DS.

Then there are the two different classes of DS that are mentioned before, design research and researching design. This thesis clearly follows the lines of design research. It is focused on specific application context, which is the testing of the company's software. It will be influenced by the constraints of the company's software; the results will be specific for this exact implementation and require additional research for generalization. Outcomes of this research focus on the artifact, which is the testing tool, and general improvement. It is also mainly focused on the fields from which one of them is software engineering, which is the field of this company. Based on these justifications mentioned in this chapter, it is decided that this thesis uses design science as its research method.

3.3 Design science research process

For application of design science in this thesis, the guidelines for DS research from Hevner are used as a baseline. These can be seen in Table 3, and are further explained in (Hevner et al., 2004). These guidelines are meant to help apply DS into the research process and act as a framework for research. Peffers has a slightly different approach for DS research and a methodology of his own, which has six activities. These are: problem identification and motivation, define the objectives for a solution, design and development, demonstration, evaluation, and communication. In this methodology, the focus is that there are multiple entry points to the research process depending on what kind of case is the topic of the research. It tries to meet three objectives: provide a nominal process for DS research, be built on prior research and provide a mental model for structure. (Peffers et al., 2007) This methodology is an improvement to his prior research to DS research process that handles the same topic (Peffers et al., 2006). Salvatore also thinks that there are six activities to research process (Salvatore T. March and Veda C. Storey, 2008). Hevner argues that when working on DS research, three cycles must be present. Relevance cycle initiates the research with context

Table 3. Design-science research guidelines (Hevner et al., 2004)

Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.

Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

and provides opportunities, problems, and acceptance criteria. Rigor cycle provides past knowledge to the project from knowledge base, which contains expertise, experiences, existing artifacts, and processes that are found and define application domain. It is based on researchers' skill to select and apply proper theories and processes to create an artifact and contribute to the knowledge base. Design cycle is the heart of the project, which iterates between building of the artifact, its evaluation and feedback to design it further. It requires information from the other cycles to do the actual work of the project. (A. Hevner and Chatterjee, 2010) Figure 1 illustrates these cycles. There should be balance between the artifact creation and evaluation based on the other cycles and understanding of the dependencies of the cycles, even if the design becomes quite independent.

Next, we apply the seven guidelines presented in Table 3 to this project and explain them as described in (Hevner et al., 2004). Guideline 1 says that the research must produce a viable artifact. In this case it would be the selected testing tool for the company's software. It is purposeful, provides utility to a specific problem, it is coequal with business needs, and it is complete when it satisfies the requirements and constraints of the problem it was meant to solve. It can be argued also that it is an innovation of some level that defines ideas, practices,

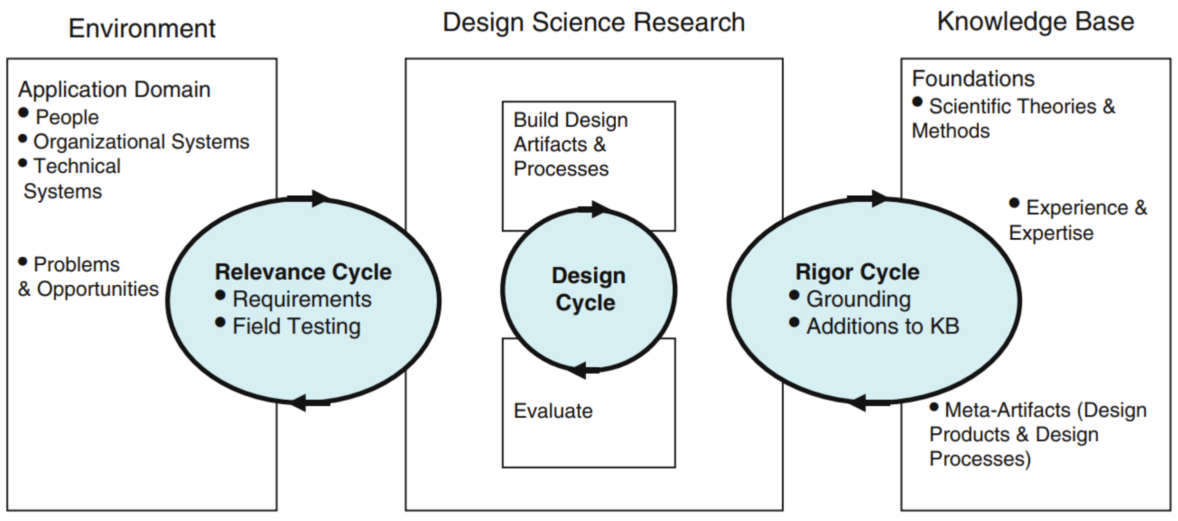


Figure 1. Design science research cycles (A. Hevner and Chatterjee, 2010)

technical capabilities, and products, but that depends more on the company. Guideline 2 defines problem relevance. The problem in this case is the improvement and lack of testing, which is a problem of specific domain. It can also be defined as a difference of current and goal state of the system as there either is testing present with the selected tool or there is not. Guideline 3 describes evaluation of the artifact, and more detailed discussion of the artifact and its evaluation comes later in this thesis. Evaluation must be done using well-executed evaluation methods and it is crucial that it is done thoroughly. Requirements and constraints, which against the artifact is evaluated, comes from the business. Evaluation includes integration of the artifact to the business infrastructure, and it is an iterative process to improve through feedback. Information technology artifacts can often be evaluated through functionality, completeness, consistency, accuracy, performance, reliability, usability, fit to the organization and any other relevant quality attributes. It is important to select the appropriate evaluation methods, depending on the nature of the artifact. Design evaluation methods can be categorized as following: observational, analytical, experimental, testing, and descriptive. As the artifact is a software testing tool, the closest methods for evaluation would be analytical for performance evaluation and experimental to test its usage. Possibility is also a descriptive informed argument that uses information from the knowledge base. Guideline 4 requires the research to make clear and verifiable contributions. Contribution type must be either artifact, foundation, or methodology related. Here the contribution is the artifact created and solving the problem to the company. Guideline 5 is about rigor. It means that proper research is done, and usage of knowledge base is effective and proper. The evaluation of the artifact is also done rigorously and with proper techniques. Guideline 6 is about the iterative process of the DS research. This can be seen at the beginning of the implementation process where search for the proper testing tool is ongoing. Pool of tools is found, checked against requirements and a sub pool of tools is created. This iterative process continues until one is found. Guideline 7 defines the communication of the research. It must be clear to all audiences but contain enough detail that technical people can replicate it. For other audience it must contain enough knowledge so that they can apply this artifact within specific context.

3.4 Research artifact

The research artifact is the most important part of DS research because it requires it to be produced, as mentioned in the first guideline of DS research (Hevner et al., 2004). Since it is so important, here the artifact is examined more carefully. According to (Peffers et al., 2012) there are several artifact types that can be classified such as; algorithm, construct, framework, instantiation, method and model. There are also other opinions about the possible classes of artifacts. Cleven in his artifact evaluation framework also mentions theory as an artifact type, while leaving out algorithm and framework (Cleven et al., 2009). Salvatore and Veda speak from information technology perspective that artifacts are defined as constructs, models, methods and instantiations (Salvatore T. March and Veda C. Storey, 2008). Venable mentions in his framework for design science evaluation that artifacts can also be classified for either product or process and technical or socio-technical artifacts (Venable et al., 2012). Alturki mentions that information technology artifacts support organizations to achieve goals (Alturki et al., 2012). Lee shows us an example by introducing a grammar-based method and a software prototype, demonstrates their use and then evaluates them (Lee et al., 2008).

In this project the artifact is the GUI testing tool, that is selected by narrowing down a list of available tools based on requirements and constraints. As a tool and a software product, it limits the possible classifications for this artifact. Possibilities are algorithm, construct, framework, instantiation, method, model, and theory. From product or process, this is a product because the artifact is the tool which is a software product. And from technical or socio-technical, this is a technical artifact because there is no need for social aspect for a testing tool in this research. Considering these seven possible artifact types for a testing tool artifact, algorithm, framework, and theory can be cut right away as they do not fit. For the rest of them basing on Cleven's and Peffers' descriptions, testing tool is not something that describes approach or process with logical instructions like an algorithm, framework would be a meta-model that is at a larger scale than just one tool, and theory would be describing cause and effect relations. Testing tool is also not a model because it does not represent reality with formal notion language or show relations between construct elements. Method is described as actionable instructions and representation of algorithms and proceedings for solution. Testing tool does not fit well in these constraints, so what is left is construct and instantiation. Construct is an assertion, concept, or syntax from assertions, set of statements or other concepts and offers syntax and semantic for a domain to describe the problem. Instantiation is a concrete realization of a construct, model or method that demonstrates it and allows researchers to test their theories in the conditions of the real world. (Cleven et al., 2009; Peffers et al., 2012) Based on these descriptions the software testing tool artifact is a construct if it were to be researched alone. It becomes an instantiation when it gets implemented to the company's software product, as then a construct is in use in real world conditions. The artifact will be treated as an instantiation going forward.

3.5 Evaluation

Evaluation is an important part of the DS research process. It is mentioned as a third guideline for DS research and there are multiple different ways to evaluate the research artifact, depending what it is and even the DS research process itself (Hevner et al., 2004).

Table 4 shows different evaluation methods. From the table there are many evaluation methods that can fit in this use case and many that cannot. For this project all five evaluation methods have possibilities to evaluate parts of the testing tool. This whole project is a case study for a company, testing tool can be analyzed all four ways, it can be experimented through simulation, the tool can be partly black box and white box tested and it can be

Table 4. Design evaluation methods (Hevner et al., 2004)

Evaluation method	Sub method and description

1. Observational	Case Study: Study artifact in depth in business environment
	Field Study: Monitor use of artifact in multiple projects
2. Analytical	Static Analysis: Examine structure of artifact for static qualities (e.g., complexity)
	Architecture Analysis: Study fit of artifact into technical IS architecture
	Optimization: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behavior
	Dynamic Analysis: Study artifact in use for dynamic qualities (e.g., performance
3. Experimental	Controlled Experiment: Study artifact in controlled environment for qualities (e.g., usability)
	Simulation - Execute artifact with artificial data
4. Testing	Functional (Black Box) Testing: Execute artifact interfaces to discover failures and identify defects
	Structural (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation
5. Descriptive	Informed Argument: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact’s utility
	Scenarios: Construct detailed scenarios around the artifact to demonstrate its utility

described through informed arguments and experimented with testing scenarios. Peffers expands this list with his listing of evaluation method types. Some of them are same or similar as in Table 4 like case study and some of them are completely different like prototype. These can be seen in Table 5.

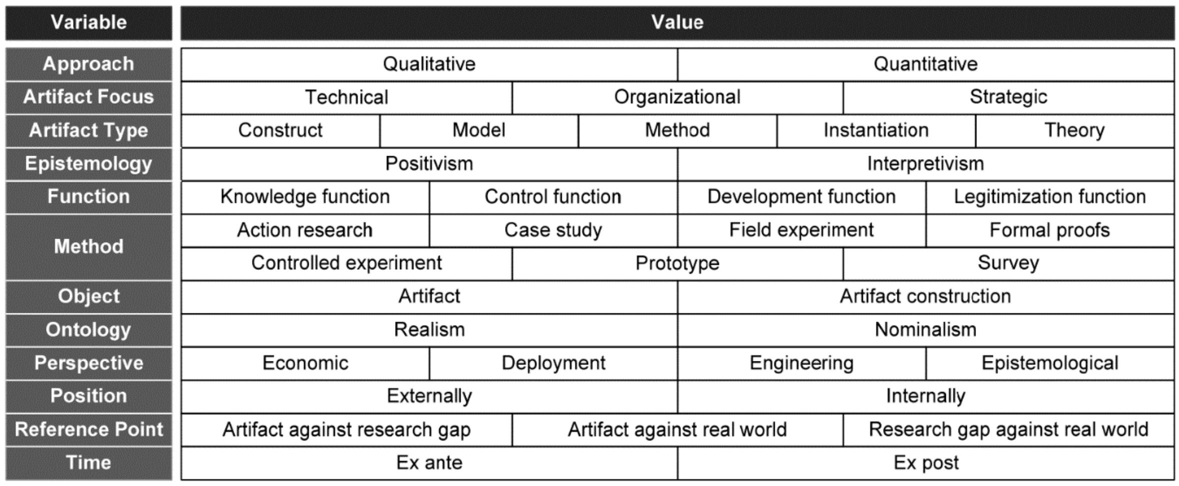
Table 5. Evaluation method types (Peffers et al., 2012)

Evaluation method	Description
Logical Argument	An argument with face validity.
Expert Evaluation	Assessment of an artifact by one or more experts (e.g., Delphi study).
Technical Experiment	A performance evaluation of an algorithm implementation using real-world data, synthetic data, or no data, designed to evaluate the technical performance, rather than its performance in relation to the real world.
Subject-based Experiment	A test involving subjects to evaluate whether an assertion is true.
Action Research	Use of an artifact in a real-world situation as part of a research intervention, evaluating its effect on the real-world situation.
Prototype	Implementation of an artifact aimed at demonstrating the utility or suitability of the artifact.

Case Study	Application of an artifact to a real-world situation, evaluating its effect on the real-world situation.
Illustrative Scenario	Application of an artifact to a synthetic or real-world situation aimed at illustrating suitability or utility of the artifact.

These evaluation methods also have some points that are applicable to this project. Some of them require a specific context or perspective to be effective. Expert evaluation requires suitable person to evaluate the artifact and illustrative scenario requires the project focus to be in the artifact's suitability or utility rather than some other metric like performance. According to Peffers' findings the most common evaluation method for instantiation artifact was technical experiment, but the choice of evaluation method varies case by case basis (Peffers et al., 2012). For this project possible evaluation methods from Table 5 are logical argument, technical experiment, action research, prototype, case study and illustrative scenario. Case study, experiments and arguments were explained previously. Subject-based experiment is not possible as there are no subjects and expert evaluation would require an expert to give evaluation. Illustrative scenario would require the focus of this project to be more towards the utility point of view and action research would require research intervention as a context. What is left is the prototype and it would fit nicely to the context of this project. Testing tool instantiation artifact is implemented as a prototype to the company's real-world software and its effects are evaluated. Difference between prototype and case study is that case study also evaluates artifacts effect to the environment and not just its utility like prototype does (Peffers et al., 2012). What can be seen here is that the point of view in the project matters.

Salvatore and Veda point out that constructs, models, and methods are evaluated by their ability to improve performance. While instantiations or implementations are evaluated by their effectiveness and efficiency in their given tasks performance. (Salvatore T. March and Veda C. Storey, 2008) Cleven has made a framework for evaluating DS research artifacts, this can be seen in Figure 2. This comprehensive process goes through each phase of the research and is with it from start to finish, as evaluation should not be an isolated process. (Cleven et al., 2009) Fitting this project to the framework is quite straight forward as some of these variable values do not have much room for unclear answers. For approach this project is more to the qualitative side as not everything is solely based on numbers. More emphasis on features and user opinions for the testing tool. Focus of the artifact is technical and

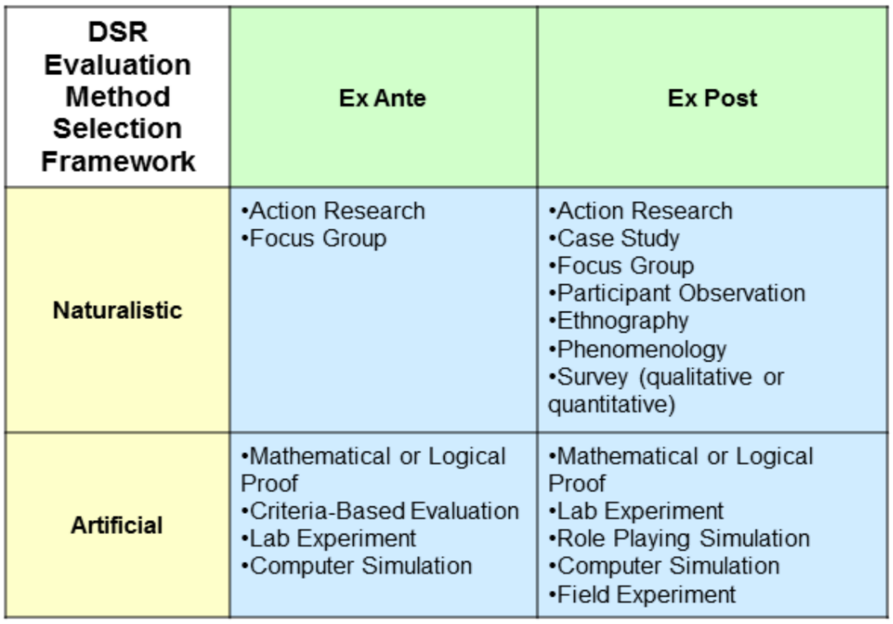


Variable	Value				
Approach	Qualitative	Quantitative			
Artifact Focus	Technical	Organizational	Strategic		
Artifact Type	Construct	Model	Method	Instantiation	Theory
Epistemology	Positivism	Interpretivism			
Function	Knowledge function	Control function	Development function	Legitimization function	
Method	Action research	Case study	Field experiment	Formal proofs	
	Controlled experiment	Prototype	Survey		
Object	Artifact	Artifact construction			
Ontology	Realism	Nominalism			
Perspective	Economic	Deployment	Engineering	Epistemological	
Position	Externally	Internally			
Reference Point	Artifact against research gap	Artifact against real world	Research gap against real world		
Time	Ex ante	Ex post			

Figure 2. Variables and values for evaluation of DS research artifacts (Cleven et al., 2009)

its type is an instantiation as it has been mentioned before. Epistemology falls towards interpretivism because the artifact is a testing tool and many tools have similar features, so in the end it comes to personal preference at some point. It is mentioned that the four different functions are closely interweaved, so it could be difficult to choose one (Cleven et al., 2009). Partly it is knowledge function as the findings of this project may steer decisions about testing and the usage or change of the tested tool. A close one is controlling function as knowledge from evaluation is used for control and it can be examined whether the predefined criteria is fulfilled. Third possibility is the development function that is based on the insights of previous functions. This allows enhancing possibility of the artifact and facilitates discourse between stakeholders. Still the closest one would be control function for the criteria examination. Discussion about methods can be found in previous section. Here are also two possibilities that vary slightly depending on their explanations, case study and prototype. Example for prototype evaluation can be found in (Lee et al., 2008) where a prototype software is evaluated using criteria that is made specifically for that implementation. For this project the same point stands that, since the evaluation takes into consideration also other points than just specific problem solution, case study is the answer. Object for this research is artifact since the testing tool is already made and ontology is reality because the program exists. The perspective is slightly economic as successful implementation will save company money, but it is more like deployment since the tool is selected based on previously presented criteria. Position in external as the testing tool artifact is made by a third party. Reference point is clearly artifact against the real world as the implementation happens by a real tool to a real software product. Time is not so clear as the tool list is gathered and narrowed down before choosing and implementing one, ex ante. Some of the evaluation is only possible when the selected tool is implemented, ex post. Also, the evaluation should be a constant iterative process that carries through the research process, but since the final evaluation comes in the implementation phase this would be ex post.

Venable has made a framework for evaluating design science research as a whole, since there are little guidance to performing DS research and evaluation is important part of the process (Venable et al., 2012). The result of this proposed framework for method selection can be seen in Figure 3. Based on the previous discussion about the evaluation methods, this



DSR Evaluation Method Selection Framework	Ex Ante	Ex Post
Naturalistic	•Action Research •Focus Group	•Action Research •Case Study •Focus Group •Participant Observation •Ethnography •Phenomenology •Survey (qualitative or quantitative)
Artificial	•Mathematical or Logical Proof •Criteria-Based Evaluation •Lab Experiment •Computer Simulation	•Mathematical or Logical Proof •Lab Experiment •Role Playing Simulation •Computer Simulation •Field Experiment

Figure 3. DS research evaluation method selection framework (Venable et al., 2012)

research places itself to the square with the case study in Figure 3. This project is naturalistic because everything happens in the real world with a real tool and real people, and to the ex-post square as final evaluation happens with the tool selected and implemented to the system. There are parts in this research that would fit in the other squares in the Figure 3, as nothing fits just to a one evaluation method clearly. When considering the results of putting this research topic through these presented evaluation frameworks, the result that comes out is complicated. There are bits and pieces that could partly fit in every single category and many of the methods share some of their features and constraints. After all this, the decided main evaluation methodology is case study followed closely by prototype, where case study basically expands prototype to the outside environment.

4 Implementation

This chapter goes through the practical part of the thesis containing the listing of tools from the internet and their filtering to few final candidates based on constraints from the company and the theory that is learned in the previous chapters. After the filtering, final candidates are tested by installing them and creating a small test with them to learn about their use. The selected tool is then implemented properly and used with testing methodologies that are best suited for the selected tool and target of the implementation. At the end the ready solution is described.

4.1 Choosing testing tool

For the selection process of the testing tool a table is created based on the tool table from the comparative GUI testing tool study by (Sabev and Grigorova, 2017). This table contains general information of the tool that is used for the filtering process. Fields in this table are name, vendor, Uniform Resource Locator (URL), latest version, Operating system (OS), supported technologies, license, demo, update status and last update.

4.1.1 Gathering of the tools

The gathering of the tools began by looking at the sources of the study by (Sabev and Grigorova, 2017) and from there a Wikipedia article was found. This article was a listing of some testing tools and it contained some basic information about the tools ("Comparison of GUI testing tools - Wikipedia," n.d.). Some tools were taken from the study's list as well. Then the search continued by making a google search with keywords "GUI testing tools list". Ignoring advertisements, homepages of single tools and focusing on lists, the same Wikipedia article is found alongside ("35+ Best GUI Testing Tools with Complete Details [2022 LIST]," n.d.). These two lists are inspected briefly tool by tool and if the tool is linked to GUI testing, it is added to the table of tools. After these lists are done the google search results are read further, additionally ignoring any list results that do not add any new tools to the table. Other list that is found is ("58 Best Automation Testing Tools," n.d.). This list also contains pure automation tools that are not linked to the GUI in any way. Same process is used with this list, meaning that if the tool is linked to GUI testing it is added to the table of gathered tools. After these three lists the table contained 55 tools. This table can be found as Appendix 1.

Work continued by filling the cells of the table for these tools. Information was found from their respective websites, code repositories if the tool was open source and from official documentation if it was available. Supported technologies were filled by what could be found from the overview pages of the tool and its documentation, without going too deep into the tool's documentation. If some table cells could not be filled because the information could not be found, it was left blank. Tools were marked either updated or deprecated based on whether the tool was updated after 01/01/2020 or dead if the website for the tool was broken or could not be found at all.

4.1.2 Limiting selection

From the table of 55 tools, it needs to be reduced to a smaller number. This is done based on the requirements from the company. These requirements were decided by having a discussion with the person in charge of this project and can be seen in Table 6. These requirements

Table 6. Requirements for the testing tool.

Number	Requirement
1	JavaScript or typescript programming language.
2	Is usable with VSCode IDE.
3	Should work on modern browsers
4	Must integrate to Jenkins
5	Should have recording functionality
6	Must have a future
7	Open source

are JavaScript or typescript programming language as this is done for a Vue.js project. With this, there is no need to learn any additional languages. The testing tool should be usable with VSCode Integrated Development Environment (IDE) as this is the one that the company uses and has all internal materials for Vue.js development. The testing tool should work on all modern browsers, meaning Microsoft Edge, Mozilla Firefox, Google Chrome, and Safari. These browsers can be divided based on their engines as Blink-based which is part of chromium project and includes Chrome and Edge, Gecko-based which includes Firefox and WebKit-based which includes Safari. The tool should be able to work with Jenkins because the company uses that in their CI/CD pipeline. It should have a recording functionality so that other people than developers can create tests. Additionally, the tool must have a future, meaning that if the company ends up with the selected tool it does not get abandoned within a year or two. Then there is a very strong preference about the tool that it is open source, so that if something happens or is questionable the source code is visible, unless an incredibly good paid tool is found. Basically, this is treated as a firm requirement as many of the open-source tools are on par with the paid proprietary ones.

Based on these requirements the tool table can be filtered for smaller pool of tools. First dead and deprecated tools are filtered out from the update status column. Then proprietary is filtered out from the license column and from vendor column only open source is selected. Last step is to filter column supported languages or technologies to show only the tools that can use JavaScript. This leaves us with 9 tools left that are updated, open source and use JavaScript.

These 9 tools are Appium, Cypress, Gauge with Taiko, Nightwatch, Opentest, Oxygen, Playwright, Selenium and WebdriverIO. Gauge is a test automation framework and Taiko is a tool for web application testing and they are recommended to be used together by their websites. Taking a proper look at the websites of these tools reveals that Appium is meant mainly for mobile application testing and is therefore rejected. Out of 8 tools Playwright is the only tool that seems to fulfill all the requirements. It has a proper recording feature, it works with VSCode, can use JavaScript, can integrate with Jenkins, and can run on all browsers. It is also backed by Microsoft so there is reasonable belief that this tool is not abandoned for a while. Cypress and Gauge with Taiko does not support Safari and the rest require the tool to be used with a Mac. This is because the browser driver for Safari is maintained by Apple, and it comes with Safari browser for Macs. Oxygen and Selenium are the only other tools that have a proper recording feature, but they have their own IDEs and Selenium has a browser addon for Chrome, Firefox, and Edge. WebdriverIO can utilize Read Evaluate Print Loop (REPL) functionality to drive the browser from the command line. Taiko can utilize

REPL the same way to drive the browser but can afterwards generate the REPL session as code with a command. The use of REPL can be arguably bad as command line has no quality-of-life features like auto filling commands compared to a modern IDE. All these tools can be integrated with Jenkins with varying difficulty, so it will not be a problem.

While taking a closer look at Opentest it can be seen that it is a usable tool with many features combining several other tools like Appium and Selenium. Created in 2016 and still being in version 1.x.x. while it is written by a single user with rather long update cycles, latest being two years. Even if this tool would be perfectly usable, the fact that it is created and maintained by one person only and updates take this long makes this tool hard to trust. In this situation where tool is searched for a company to use for long term, it requires trust and when a tool is completely under one user's actions, it is too much risk to take. Even if the tool is sponsored by McDonalds and started from there. For these reasons Opentest is also rejected.

The Vue.js official website ("Testing | Vue.js," n.d.) also recommends Cypress, Playwright and Nightwatch for end-to-end testing. Keeping these things in mind, the following tools are tested further to gain a better opinion and showcase them to the company. These 7 tools are Cypress, Gauge with Taiko, Nightwatch, Oxygen, Playwright, Selenium and WebdriverIO with Playwright being most promising of them all based on requirements. Even though all of these tools do not cover all of the requirements, mainly recording, they are still kept here to gain better general understanding what tools like these can offer. They can also be compared to each other as selecting the best tool right here offers little value without comparison.

4.1.3 Testing and notes about the final candidates

Testing of the remaining tools is done with the actual piece of software from the company that is written in Vue.js. First the tool is installed with everything that it needs. This means possible browser drivers that might not be included in the tool and have to be installed separately. Then the examples are executed to make sure that everything works, and this is done with all browsers that are supported by the tool in Windows. Then a test is created to the company software which goes as follows. First the tool navigates to the site, opens a dynamic sidebar, creates new event, adds information to the event and saves it, finds the created event, and edits it, finds this event again and deletes it. This test is created with each tool and the results are discussed with the company.

After discussions with the company, the seven different tools have been reduced to two choices. The final selection will be between Nightwatch and WebdriverIO. There were two qualification rounds. In the first one, each tool was introduced in general. This included discussion about features, usability, documentation, configuration, support, installation, integrability and demo runs of the tool's examples and the created test that is described in the previous section. In this first round Gauge with Taiko, Oxygen, and Selenium were dropped. In the second round a closer look into the company's CI/CD pipeline is taken and based on this Cypress and Playwright is dropped for being incompatible. This lefts Nightwatch and WebdriverIO for two final candidates. The following comments are the results of the discussion with the company.

Gauge with Taiko was eliminated for several reasons. The biggest one being its marketed feature of writing tests in markdown that is easy to read. While it could have been okay if only markdown was used, the test steps that the markdown uses need to be coded too. This effectively means that everything must be done twice, once with JavaScript for the logic and once with markdown to drive that logic. This could rapidly devolve to just writing everything in JavaScript and then minimal markdown to run the tests, but why use this tool at all then? In developer centric environment where the code is readable for most people, this just creates too much extra work. Not to mention that learning a brand-new syntax is something that is not preferred. The support for this solution is community based with quite small community and it seems to be in slight hibernation. This is a risk for long term use and the REPL feature in Taiko just seems laborious with not much use. Even if the REPL is used for recording tests in this, it might be faster to just write code in IDE with all quality-of-life features than drive browser with perfect syntax in command line and record it.

Oxygens biggest turn of was its own IDE. It is not preferred to add yet another development environment to the mix. While it has a rather good recording feature with browser extension that allows to add assertions and waits in the middle of the recording, the generated code is not too readable as it includes all possible locators as comments next to every segment of generated code. Documentation is not as good as the other candidates as things were little bit harder to find. It has smaller community and community pages that are meant for help did not have much recent activity. It just did not convince.

Selenium was close to not being one of the final candidates but was left there to give a better context for GUI testing tools. It was dropped for being too basic overall, when compared to other tools. Fact is that many other tools are based on Selenium and that brings its own advantages and disadvantages. The base Selenium is basic with limited features, and it will not develop too quickly because so many other tools depend on it. If there would be unlimited time and one or two coders experienced with Selenium, it would be the dream tool as it is a good base to build on. But the company does not have too much extra time or experience with selenium, so this tool is just too much work. It has great documentation and instructions to develop it further with a large community from nearly twenty years. It has its own IDE as a browser extension for Google chrome, Firefox and Microsoft Edge which is used for recording tests. The default .side files are not properly human readable, but it can export recorded test for other languages like JavaScript. In short it just requires too much work, and the own IDE is not preferred but otherwise looks good.

For the second qualification round a closer look is taken on how precisely these tools integrate with company’s development pipeline. Looking through the documentation for each tool, it is revealed that Nightwatch and WebdriverIO integrate to Jenkins with node.js while Playwright and Cypress would require docker containers to run their images in. Problem is that the company’s current development pipeline does not work with docker. Their infrastructure is mostly based on FreeBSD and the servers are made similarly to make their management easier, as there is roughly a hundred of them. While there are few Linux based servers and if there becomes a situation where there absolutely must be a Linux server, it can be done, it is strongly not preferred direction.

Cypress is different compared to other tools as it opens its own app in the browser which is tested on. Because it runs inside the browser it can see everything that the browser does and inform it to the developer. It is very developer centric tool that is simple to use and designed to developers. It has a paid dashboard service that gives it cloud functionality but that is completely optional and free base tool is capable of everything that is required to use it properly. The documentation is great, and it has the possibility to record video of test runs. It has also the possibility to test individual components and it is recommended by the Vue.js website. The only negative is that it has no ability to record tests and it does not support the Safari browser. Final decision is based on the fact that it is not compatible with the company’s CI/CD pipeline and is therefore dropped. Otherwise, it would have been a great tool.

Playwright could have been the dream tool for this case. It has great integration with VSCode which is the desired development environment, it has good test recording feature that generates readable code, it is backed by Microsoft so there would be minimal risk in trusting this tool and it can drive all browsers. It drives the base browser engines and is based on the fact that if the test runs on the base browser engine it can run on the browser that is developed on that engine. The documentation is great, it has good integrability, good default reporting, possibility to record video of the test run and good readable syntax. Only problem being the fact that it requires a docker container to run in CI/CD pipeline and that is not possible for the company's current infrastructure. For solely this reason it is dropped.

Then for the two final candidates. Nightwatch is a little bit newer tool that is still growing. It has good core functionality and quite fast development rate. It is maintained and developed by a small core team with still rather small community that is still growing. It has good documentation and default reporting. Integrability to other reporters is good and it integrates with Jenkins through node.js. This tool is developer focused with good readable syntax and simple to use as there is minimal configuration needed to get the tool running thanks to installer that takes care of most of the work. There is no recording functionality for test runs or for creating tests and the support is fully community-based. It is a complete package for a testing solution that seems to have a quite promising future.

Finally, WebdriverIO. This tool is well matured with a dedicated team of people who maintain the tool and technical steering committee that leads the tool's development. It has a large community that can help to solve possible problems and there is also a possibility to reach out to a paid expert for help, at least at the time of writing. The tool itself is very modular and needs to be built and configured for your needs, but this is handled well by the installer. The modularity comes in the form of different ready-made services that can be easily added by just adding line to the configuration file and installing the right package. This means that the tool includes only the features that you need and has therefore good performance. It has good integrability and integrates to Jenkins through node.js. The syntax for the code itself is clear and code is readable. It has good documentation and the possibility to also build your own custom services that you can add, thanks to the access to all the hooks in the configuration file. It does not have a feature for recording new tests, but it can record a video from test runs, with the right added service. It also has a REPL feature that has no code generation, so its usefulness is questionable. If nothing else, this tool seems to be a reliable choice, for its long history and maturity.

4.1.4 Final choice

Both tools WebdriverIO and Nightwatch were implemented to the company's Jenkins pipeline to make sure that there would not be any problems with the tool. Testing included two reporters for JUnit reports and Hyper Text Markup Language (HTML) reports, taking screenshots on failure and running the previously made test in the pipeline. Application programming interface (API) calls were mocked, so everything could run locally in Jenkins.

While discussing with the company it became apparent that both of these tools were extremely similar. Going through the same points about these tools as in the previous section, there were only few clear differences in functionality and for the most part same things could be done with both tools, just a little bit differently. One clear difference between them is that Nightwatch is aimed to be a complete testing tool with a solid core package that includes all basic functionality like reporters, while WebdriverIO is more modular in design and all features are different services that are added to it. Both tools also allow creating custom -services in WebdriverIO and -plugins in Nightwatch, -reporters, -commands and -assertions. They also allow access to different hooks that can be used for configuration. Another difference that came up was their configuration files. While they also contained mostly same things, WebdriverIO's configuration file contained all possible options that were not in use as comments while Nightwatch's file only contained what was in use and the rest needed to be found from their documentation. These two differences are a plus for WebdriverIO but nothing too major.

After the tools are checked and both are still valid options the eye turns to their backgrounds. Since this is a company that is searching for a testing solution, background can have a great importance for the future and trustworthiness of the tool, because after everything is done, it is expected to be used for many years. For this end, WebdriverIO is part of the Open JS Foundation and that is a project of the Linux Foundation. This was very reassuring for the company and basically means that there is no chance that this tool would just suddenly die out of nowhere. As for Nightwatch it was made originally by Pine View Software and later acquired by BrowserStack in December 2021. This was a slight source of uneasiness as BrowserStack is a private company and while they intend to keep Nightwatch open-source and create a committee to drive its development further and keep it on track, there is a possibility that it could change anytime ("BrowserStack blog," 2021).

Based on these results, the final choice is WebdriverIO, and it comes mostly from the background of the tool. From company’s comments, it showed more trustworthiness, reliability, and maturity. Also, the small pluses were the modularity in design and the structure and presentation of the configuration file as otherwise the tools were very similar.

4.2 Choosing testing methodologies, strategies, and good habits

Testing theory is further explained in the 2. chapter which includes explanations about testing methodologies, -automation, -strategy, and GUI testing. Based those findings, proper methodologies are chosen for this implementation, use of the chosen testing tool and overall GUI testing.

Target of implementation is a single page application tool, that is used to forecast something what it is implemented for. This can be for example the usage of electricity. To the front page the application loads a list of already made chart names from the server. By clicking any of these names, the page loads an amChart, loads settings for it and fills it with appropriate data from the server. On top of the chart there are buttons to move eight or twenty-four hours to forwards or backwards. On the side of the page there is a sidebar that contains a list of stoppages, that can be added, edited, or deleted. These stoppages contain two dates, estimate number and a comment. While doing any of these actions the stoppage list is covered with the action’s own component and the whole assembly can be hidden at any time by clicking the sidebar again.

As it is mentioned before, GUI testing is difficult for multiple reasons and some of them apply also here. First is the dynamic content. There is dynamic content that move in and out of vision, animations that play during loading and components that are created during runtime. This immediately limits the use of more traditional methods and strategies for this use case (Kandil et al., 2012). Since the target of implementation is rather simple one page application, there is no need to check every link as the link on the front page just decides what data is loaded while the page itself remains the same. The link could be named anything, and the GUI would still work, just the data would not be loaded for the charts and the charts would be empty. This could however be great method for larger application that has multiple pages (Di Lucca and Fasolino, 2006).

There were also mentions that both manual and automatic testing have their places and it would apply also here (Kasurinen et al., 2010). The amChart at the center of the page should not be tested automatically as it difficult to interact with an automatic testing tool. Because the chart is made of hundreds of automatically generated elements it can change easily and any automatic test would immediately break. The chart is also made by a third party so it should be considered if there is any reason to test it exhaustively as not much can be fixed internally, other than configuration. Testing of elements like these should be done manually to minimize the work required (S. Berner et al., 2005). For automatic testing the best that can be done without writing tests again each time is to just check if it exists on the page.

Other parts of this application can be tested automatically, and smoke tests are a good place to start. This means the testing of basic functionality broadly and it reveals if there are any breaking bugs (Pettichord et al., 2011). In this case it would be the creation, editing and deletion of a stoppage and moving forwards and backwards with the buttons above the chart. These smoke tests can then be improved to be more thorough by adapting equivalence partitioning and boundary value analysis to them. For example, in this application for checking how the lists for charts and stoppages react when there are no entries or there are a hundred. These two testing methods should be also used on each input field to test the application as thoroughly as possible from the GUI side. It is also important to test for bad API calls and how the GUI responds to them as it is a web application. These can be tested easily by mocking the calls and return what the testing requires.

If the application would have multiple pages, a page coverage and link coverage are good candidates for testing completion criteria, but this should also be considered case by case basis (Di Lucca and Fasolino, 2006). For a smaller and simpler application like this, going through all functionality is reasonable but that is not always the case. Generally good targets for automated GUI testing are routine operations, frequently used functions, operations that require accuracy and long end to end scripts (Zhyhulin et al., 2022). The larger the testable software is, the more complex its testing becomes and therefore tests can break more easily. Other reason for breaking tests is unstable software (C. Persson and N. Yilmazturk, 2004).

Then for the tool, as it does not really care about testing methods, but there are things that can be done to use it more effectively. Overall scripts should be short and simple, focusing on one thing and contain three clear sections for preparing, executing, and cleaning (Singhera et al., 2008). This makes the scripts easier to maintain and control. For each different running environment there should be a separate configuration file as it is much easier to call a separate file rather than always change something back and forth in the file. WebdriverIO allows you to access different hooks that can be called, and these should be used to reduce the repetitive tasks that need to be done. For example, before each test go to specific URL. Each test should start from a neutral point so that only one test fails. If tests are dependent of each other, after one failure all subsequent tests also fail. Using complex selectors for the elements, like full XPath, should be avoided as they are easy point of failure, because if any element before the selected one changes then the selector must be checked as it could have changed too. Using unique identifiers for elements is recommended as the tool can find the element just by that identifier and it will not care about other elements between them (Bruns et al., 2009; Singhera et al., 2008). Meaning that anything else on the page can be changed and the element can still be found.

In conclusion not everything should be automated, even if it would be possible. GUI testing is a difficult part to automate, so the starting point should be simple smoke tests that cover basic functionality. Testing overall should start from simple scripts that each cover one thing and later expand the testing effort for more complex actions. Tests should also not be dependent of each other. But if tests start to get too complex, their maintenance becomes an issue, and that line is for the company to decide and find out. For larger web applications link- and page-coverage are good option for completion criteria as the more traditional ones might not fit well to the modern wed applications. For the tool the most important part is to use good selectors to find the elements, and this comes back to the fact that the software itself must be made testable.

4.3 Implementing a solution

Implementation starts from installing the tool and adding it to a project. Instructions from the WebdriverIO documentation are followed (“WebdriverIO,” n.d.). WebdriverIO has a nice installer for command line use that handles everything nicely and user just chooses some basic functionality. Command ‘npm init wdio’ was run and chosen answers during installation can be seen from the Table 7. After the installation, the autogenerated example tests were executed with command ‘npm run wdio’. This would confirm that the tool is installed correctly.

Table 7. WebdriverIO installer questions and picked answers.

Question	Picked choice
Where is your automation backend located?	on my local machine (default)
Which framework do you want to use?	mocha (default)
Do you want to use a compiler?	no (default)
Where are your test specs located?	./test/specs/**/*.js (default)
Do you want autogenerated test files?	yes
Do you want to use page objects?	no
Which reporter do you want to use?	spec (default)
Do you want to add a plugin to your test setup?	-
do you want to add a service to your test setup?	selenium-standalone
What is the base url?	http://localhost (default)
Do you want to run 'npm install'	Yes (default)

Selenium-standalone service runs the browsers on a selenium server automatically without the need of any additional configuration, so it is a simple solution. This was the baseline and from here the building of the solution would begin. All commands used in this process can be found in the Appendix 2.

After the base WebdriverIO was tested on local machine, it was tried on the company Jenkins server. The existing Jenkins file was cleaned from all extra like email notification, to make it more feasible to run repeatedly in order to test it. The full Jenkins file can be found as Appendix 3. A new pipeline job for testing WebdriverIO was created to Jenkins and it would pull latest changes from the repository and then execute the Jenkins file. The application was pushed to repository and the pipeline would be run from the Jenkins dashboard. First run would just return error from the selenium server when testing stage began. Reason for this appeared to be the fact that the company Jenkins runs on FreeBSD operating system and the WebdriverIO selenium-standalone-service would just not support it. This means that the simple one service solution is out and separate services for the browsers would be used. Discussing with the company it was decided that just chrome browser would be added to the Jenkins for now as browsers apparently cause much security headaches for the managing side. Also, the fact came up that the company Jenkins is basically closed and the easiest way to test the application was to run it locally in Jenkins.

Once the base tool was ready, changes were made to the target of implementation which is the forecast application. This application is described at the beginning of section 4.2. First change was the addition of id-parameters for elements, most importantly for all buttons and inputs that did not yet have one, so that WebdriverIO can more easily find them. Then a mock-up was created for the application so the GUI could be tested without the connection to the backend. This was handled with Mock Service Worker, that was found after a brief search from the internet. It is a simple and clean opensource tool for mocking so there was no objection from the company. Following its instructions from the homepage ("Mock Service Worker," n.d.), a basic implementation was created. A simple mock-up was made by copying the first responses from the server to the API calls, just enough so that it would work. Then the same test that included addition, editing and deletion of stoppage was created.

Since selenium-standalone service does not work, each browser requires their own driver manually installed. For Chrome and Firefox browsers there is a ready service for WebdriverIO that can be just installed via command line and then added in the configuration file to the services section. All changes towards the final configuration file can be found as Appendix 4. At this point separate configuration files were made for running WebdriverIO locally and in Jenkins as making constant changes to a one file is just extra work and distracting. A new launch script was also added to the package.json that used the local configuration file. These services handle everything and no additional configuration is required for them to work. The service for Microsoft Edge is outdated, so for that browser to work its driver needs to be downloaded and added manually. With these added by installing wdio-chromedriver-service and the wdio-geckodriver-service, the test for the application could be run locally with all the browsers. It was also discovered that chrome driver could also run edge browser if there was no Edge driver present, most likely because they are both based on chromium.

This was also tested in Jenkins, and it would not work. Chrome driver would not start at all, Firefox driver would start but it could not connect to anything as there were no Firefox browser installed in Jenkins and Microsoft Edge was not tried as the driver was not installed in Jenkins. Firefox was also disabled, and all further testing was only done with chrome. As there was clearly something wrong with the driver service, a mac chrome driver was added manually to the project and a path was added to the chrome-driver-service in configuration file. Since the company Jenkins runs on FreeBSD a mac driver is the closest one, compared to Windows and Linux versions. After finding the right path to the driver in few tries, WebdriverIO would start and fail since it could not connect to the testable site, but it launched properly. Now the things turn to the testable application.

To run tests in Jenkins, the application should also be run in Jenkins. This is not a must, but it is preferred by the company and nearly all connections to outer world are closed in their Jenkins server. It is also a good practice since the focus of the testing is just the GUI and not the complete application. The application uses Vue-cli and it has a ready command to launch a developer server. Idea was to use this to launch the application, then run the tests and close the server for Jenkins job to finish. Problem is that when the server is running the shell only listens to the server and does not take any other commands unless interrupted and that closes the server. This would mean that either the application or the tests would run but not together. After some research to Jenkins and specifically to Jenkins file and its syntax, a parallel step is found (“Jenkins User Documentation,” n.d.).

With parallelization Jenkins is basically running multiple shells at the same time and so it can take multiple commands at the same time. To the Jenkins file under the test stage a parallel step is added. Inside the parallel step, own stages for ‘run dev server’ and for ‘run tests’ is added. Now Jenkins runs both at the same time but immediately a problem occurs. Since both start at the same time the server is not ready when tests start, and the tests fail on timeout. This can be solved by adding a sleep command in the run tests stage before the command for running the tests. This gives enough time for the server to start, and the tests run now perfectly.

Then another problem appears. The server is never closed so the Jenkins job never finishes until it times out and has to be aborted from the dashboard. When running locally, the developer server would be closed by just closing the shell that is running it or sending an interrupt to the shell, meaning ctrl + c key combination. After some more research in Jenkins a timeout function is found. This can be set to basically any step, stage, or agent. So, in the ‘run dev server’ stage a timeout block is put around the command of launching the server. After the set amount of time, it sends an interrupt signal to the shell and the server closes. Next problem from here is that now the Jenkins job always fails because the developer server is interrupted, and it counts as an abort and therefore failing the job. After some more digging in Jenkins documentation and internet generally, a possible solution emerges.

When timeout times out it sends a specific error. This error is: “org.jenkinsci.plugins.workflow.steps.FlowInterruptedException” and it can be caught, checked and if it matches, the current build can be set to be successful. Jenkins file does not accept any scripting, so it must be written inside a script step. So, inside the timeout block, there is a script step which has a try-catch block for running the developer server and catching the timeout error. If the error matches, current build is set to successful and script returns. With all these changes the Jenkins pipeline worked perfectly.

Now that the base testing works both locally and, in the pipeline, it is time to add reporting for the tests. WebdriverIO has multiple ready-made reporter services and JUnit reporter is one of them. It only requires the @wdio/junit-reporter package to be installed via command line and then the ‘junit’ needs to be added to the reporters-section of the configuration file. Documentation for the JUnit reporter in WebdriverIO website shows basic configuration that can be done with the reporter. By default, it creates a .log file but that can be changed by following its documentation and changing outputFileFormat to return an xml file. As for Jenkins, it has a JUnit plugin that publishes JUnit report to the jobs page that can be configured in the Jenkins file. To the post section of the ‘run tests’ stage inside always -condition a line is added for publishing the report. This line is: junit “location of the files” and in this case it is: junit “test_output/*.xml”. By using the wildcard * the plugin finds all .xml ending files from that folder and publishes them.

Next a HTML reporter is added for better readability and visualization. For WebdriverIO there is a ready-made service also for this purpose. For this reporter a wdio-html-nice-reporter package needs to install and then html-nice needs to be added to the reporters-section of the configuration file. To the top of the configuration file an import for the reporter needs to be done. From the documentation, an example configuration can be seen that can be copied and it defines names for report and files, path to output directory some Boolean parameters and possibility to use logger which is not used in this case. The base HTML report is just basic black and white page and has the statistics of just one test spec file. To create one uniform report, the HTML reporter allows aggregating the reports and creating a one master report. For this to work an additional configuration step is required and this involves the hooks that WebdriverIO has. To the onPrepare hook, a function for gathering the test reports is created and it has the configuration for the name and location of the master report. To the onComplete hook, a function for creating the report is added. These functions are copied from the documentation of the reporter from the WebdriverIO website. To show the HTML report in Jenkins, it has a HTML publisher plugin that is used. To the Jenkins file in post section to the same location as the JUnit report, a line is added for the HTML report. The full instructions can be found from the Jenkins HTML plugins own documentation, but basically it requires the location of the report files and the name of the report. Jenkins then publishes the HTML report so that the latest one can be found from the jobs sidebar and by going to specific build it shows the build specific report.

Immediately it was noticeable that the HTML report was not interactable. Checking browser log reveals multiple errors about blocked content. Jenkins HTML publisher plugins page's first instruction is to check the Jenkins content security policy that by default sets a response header so that it basically blocks everything that happens on runtime. With the help of Jenkins documentation and content security policy guide this problem could be solved ("CSP Reference & Examples," n.d.). By trying different options to the Jenkins console and eliminating errors one at the time from the browser console, a working solution was found. The final command for changing the response header was:"System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "sandbox allow-scripts; default-src 'self'; img-src 'self'; style-src 'unsafe-inline'; script-src 'unsafe-inline'; style-src-elem 'self'; script-src-elem 'unsafe-inline' 'self';")". Now the HTML report was fully interactable

After the base reports were working a possibility to record a video from the test execution was examined. WebdriverIO has a ready service called video reporter and it can be added by installing wdio-video-reporter package and add line: "const video = require('wdio-video-reporter');" and then add 'video' to the reporters-section of the configuration file. This worked great locally but in Jenkins there were problems. Because the video reporter works by taking screenshots after actions with selenium and then combines the screenshots to a video, it throws an error for unsupported platform. This is because the company Jenkins runs on FreeBSD operating system and selenium is not supported. Therefore, the video reporter is not a part of the final implementation.

Since video is not an option a possibility for screenshots is investigated. There is no separate service for taking screenshots and there are no specific instructions for it. WebdriverIO has a browser function for taking a screenshot browser.saveScreenshot(). It takes a file path as an argument and that can be added to an afterTest hook. To make sure that each screenshot has a unique name, a date is added to the name of the screenshot. When put inside an if statement that checks for an error, the screenshot is taken when there has been an error while running a test. When the file path for the saveScreenshot function is to the HTML reporter's html-reportscreenshots folder, the screenshots get automatically added to the HTML report for their respective test steps. Currently there is a known issue that the screenshots are attached to the next step on the report rather than to the one where it was taken, but that is a minor issue to live with. In Jenkins the screenshots would not automatically show and by looking at the browser console the reason was that the pictures were blocked by the content security policy. Since the images were base64 encoded, by adding data: to the image sources, so now the image source would be:" img-src 'self' data:;" the screenshots were visible in Jenkins' HTML report.

At this point while reading documentation for WebdriverIO it is noticed that a recorder for tests has appeared to the documentation. This is either a chrome recorder extension or a command line tool and it uses chrome developer tools recorder to record the test. It can then be exported straight to WebdriverIO test with the extension or exported as JavaScript Object Notation (JSON) file and then use the command line package to translate it to an actual test script. This feature became possible with the launch of chromium 104 in early August 2022 and the WebdriverIO specific tools first appeared in late August 2022. These features are in very early phase and still incomplete but contain the necessary basic functionality. In discussions with the company this feature is noted, and its future is looked with great interest but right now not too much trust is placed on it as it is in very early phase. The extension can be installed from the chrome web store and then a recording can be exported from the chrome developer tools recorder as a WebdriverIO test script. The generated test script is quite readable, but it requires some changes to make it better and more endurable. Especially some of the locators and assertions have to be changed but overall, it is quite usable.

Now that testing works on Jenkins and reports are generated and published correctly, the tests and the mock-up are improved for the forecast application. The current mock-up was cleaned by creating a JavaScript file for each response that default exports the data that is in the file. These files can then be imported to the file that handles API calls and used without hundreds of lines of data in the code. The searching and modifying of the data are also easier, as now everything relating to a single API call is in a one file and the code for handling the call is more concise. The API calls for changing time for the chart were properly conditioned by comparing the URL search parameters and a proper response is decided based on the results of the comparison.

As for the tests, in addition for the first one that adds, edits, and deletes a stoppage in the forecast application, two more tests are created. One for changing the time for the chart by pressing the time buttons above it and other for the recorder. The third test is the same test as the first one but created with the chrome recorder and exported with the browser extension. Then this test is minimally modified to make sure that it runs properly. The point of this is to give a comparison point for the recorded test and a from the ground up written test. Additionally, the first test is improved with better selectors that can handle slight restructuring of the page and the test is clearly cut to three steps. Before each step a before each hook makes sure that each step starts from the same point, so the steps are not dependent of each other. With these tests the basic functionality of the application is tested, and no further testing is deemed necessary.

4.4 Ready solution

The ready solution consists of the selected testing tool WebdriverIO, mock-up of the forecast application that is done with Mock Service Worker, tests created for the forecast application and their integration to Jenkins.

WebdriverIO is selected from 55 different tools with the reasons being that it cleared the requirements and when compared to Nightwatch, that was the other final candidate, it had better background and its design was more modular. It was installed with default settings, chrome-service for driving chrome browser locally and manually adding right driver file to run the browser in Jenkins. JUnit and HTML reporter were added for different reports. Separate configuration files were made for running WebdriverIO locally and in Jenkins and launch scripts for both were added to the package.json file. Additionally, to the afterTest hook a step to take screenshot on failure was added.

The Mock Service Worker was installed with default instructions from their website and necessary files were added. This includes browser.js and handlers.js -files that are in src/mocks folder. Browser.js creates a worker instance with the request handlers defined in handlers.js file and to the main.js file a starting condition was added for the worker. A command automatically creates a mockServiceWorker.js to the public-folder and then eve-rything is setup. Also, a folder for mock-up data was made in src/mocks/mockData which has a separate file for each data response that can be imported to the request handlers to help manage the data.

Two tests were made to cover this one-page forecast application. A detailed explanation about the application can be found at the beginning of section 4.2 First test executes the creation, editing and deletion of a stoppage and the second one clicks through the time switching buttons above the main chart. These tests were divided to clear steps that test just one thing and are not dependent of each other. A third test was done by recreating the first test with the recording functionality and then minimally modifying it to run properly. This is to give context between recorded and written tests.

As for the company Jenkins, it already had JUnit and HTML-publisher plugins installed. A chrome browser was added to it and content security policy was modified so that HTML reports would function. Project Jenkins file can be seen in Appendix 3. It was modified to have a test stage that had parallel stages for running the developer server and the tests. De-veloper server was inside a timeout block and tests slept a while so that the developer server could start up. After tests, results would always be published through the plugins and be easily viewed at the Jenkins dashboard.

5 Results

The results of this project can be divided to two different parts. One is the practical implementation of the testing tool for the company and the other is the produced artifact as part of the design science contribution. Description of the practical part can be found at the section 4.4 where the practical results of this project are gathered.

The evaluated artifact is WebdriverIO, a browser automation test framework for Node.js. It was implemented to a small one-page Vue.js application from the company and confirmed to work locally and in the company's Jenkins server. It is an instantiation artifact, because this study does not only focus to the tool itself but also to its implementation. The evaluation is done as a case study, as not only the tool is evaluated but also its surroundings are taken into consideration. The evaluation is done mainly against the requirements that the company set for the new testing tool and these requirements can be seen in Table 6.

The first requirement states that the tool must use JavaScript or typescript as the programming language. This is quite straight forward requirement as it either uses or does not. WebdriverIO has the possibility to use either one of them and in this project, JavaScript was used. There is not much more to say that this requirement is fulfilled, and the company had no other comments about it.

The second requirement states that the tool must be usable with VSCode IDE. WebdriverIO works like any other Node.js package so it works well with VSCode. It also means that it is not locked to any particular IDE and the company liked this possibility in case something happens, and they would need to change things. Otherwise, it works so requirement is fulfilled.

The third requirement states that the tool should work on modern browsers. This can be slightly vague, as what strictly is a modern browser? Discussing with the company it is decided that this means mainly Chrome, Firefox, Edge, and Safari browsers. Opera is mentioned in the talks, but it is not as widely used as the other mentions, and it does not have straight support in WebdriverIO. It also is a chromium-based browser so even if it is not tested specifically, if something works in Chrome and Edge, as they are also chromium-based browsers, with high probability it also works in Opera. In this project Chrome, Firefox and Edge were tested locally, and in Jenkins only Chrome. Safari was not tested as it requires a Mac environment to run and there were none available as the company uses windows-based workstations. According WebdriverIO's documentation Safari should work without any additional configuration, but it just requires a Mac because safari-driver only comes with Safari browser, and it only comes with a Mac. All the tested browsers worked well and had no issues after initial setup. In Jenkins chrome driver needed to be added and point its path manually or it would not work. This could possibly be because the company Jenkins runs on FreeBSD OS and the driver cannot decide what driver binary to use, but this is just speculation. It is also quite simple to configure the separate options for browser capabilities in configuration file. Overall, this requirement is considered fulfilled, and the company liked the results that it can handle the four big browsers, even if Safari could not be tested in practice.

The fourth requirement states that the tools must integrate to Jenkins, meaning that this tool can be used as a part of Jenkins pipeline. Since the tool is basically just a Node.js package it runs inside Jenkins and the integration part comes with the reports that can be viewed through the Jenkins dashboard. The company Jenkins already had the plugins installed for publishing JUnit and HTML reports, so WebdriverIO just needs to make sure that the created reports are in right file format and go to a place where the plugins can find them and publish them. Both HTML-nice reporter and Junit-reporter have good configuration possibilities and guides as to how to set them up properly. As long as file paths and names are the same in configuration file and Jenkins file, everything works great. For viewing HTML reports properly Jenkins content security policy had to be changed, but that was quite simple, and the Jenkins HTML plugin already mentioned it as first troubleshooting step. For the company the most important thing was that it worked, so the reports were readable in Jenkins. As this is a step that is done once and then left alone for most part, the difficulty of configuration did not matter too much. The company is still happy that all configuration happens just in the Jenkins file and the WebdriverIO configuration file if anything needs to be changed and the current configuration is quite straight forward. This requirement is fulfilled, and the company was pleased with the current results.

The fifth requirement states that the tool should have a recording functionality. This is a rather new functionality in the WebdriverIO, and this project was nearly finished without it. It relies on Chrome developer tools recorder to make the recording and then use WebdriverIO chrome recorder -plugin to export the recording as a WebdriverIO test script.

Optionally the recording can be exported as JSON-file and then translated with a command line tool. However, it is in quite early phase and does not yet support all commands, but the basics are there, and it is usable. When discussing with the company, it is noted that it exists and that it is not yet fully complete. The fact that it uses chrome recorder is great because it does not require any setup to start recording and basically everyone has it already. For example, there is the possibility that a client or salesperson can use it to create the base recording and then give it to a programmer to modify it more usable. The WebdriverIO test script that is exported is clean fully readable code, but it is not perfect and requires at least a full check on the selectors and there might be a need to add waits if there are animations on the page. In the end it is very simple to use and a “nice to have” functionality, even in the state that it currently is. With these comments the requirement is fulfilled.

The sixth requirement states that the tool must have a future. This is another slightly vague requirement as no one can tell what happens in the future, but there are hints. This tool assumably has a good future when talking about software tools. WebdriverIO is part of the Open JS Foundation and that is a Project of the Linux Foundation. Comments from the company state that this gives good credibility for the tool and when comparing to other possible tool choices, WebdriverIO is good choice. Requirement is considered fulfilled.

The seventh requirement states that the tool should be open source. Similarly, to the first requirement it either is or is not and WebdriverIO is open-source tool. The company had no other comments regarding this, and the requirement is fulfilled.

Other than the requirements, a heavily involved part for the project was the creation of the mock-up for the forecast application. It was done using Mock Service Worker and the finished product is quite good. Company liked that the mocking tool was simple to setup, use, was lightweight and it did not really care about the internal structure of the mocked application. It required very little setup and then all the mocked API calls could just be added to the one file. The mock-up also worked well locally and in Jenkins so there were no problems.

The seven requirements that were set for the testing tool are all fulfilled. Some of them might be slightly vague but in discussions with the company a common ground was found, and they all were deemed as successfully fulfilled requirements. Additionally, the created mock-up was good and really easily usable. Overall, the whole project is considered a success. This has given the company a GUI testing tool, mocking tool, their implementation to company’s development pipeline, examples, and instructions to do the same for other Vue.js projects. A good amount of knowledge about GUI testing, testing automation, testing practices and the creation of testable software, what was found and applied during this project, is also shared to the company. With this information, the company can base its future GUI testing to a stable ground.

6 Discussion

In discussion this work is compared to a few related similar works in this field. What points they share, what are their differences and if there are any other noticeable mentions. Then the evaluation of this complete work as a design science work. At the end, discussion about the possible risks regarding this work are mentioned.

Several theses could be found that do a similar implementation to a software product. One does automated testing to React native applications (Salohonka, 2020) and another integrates unit testing to legacy applications (Pöysä, 2019). Two other theses are very close to this as one implements UI testing to web application (Lahtinen, 2022) and other implements visual GUI testing to a software product (Heinonen, 2020). Common for all these are the general process and that three of these use design science as the research method. Process is similar as these are all works done for a company or organization, requirements are gained from the company, necessary tool or tools are selected and the final result is an implementation of the tool and what surrounds it. There is also mention that the thesis ends at the successful implementation and no long-term results are gathered so that can affect to the validity of the results.

To make sure that this thesis follows the principles of design science a framework for evaluation in design science (FEDS) is applied (Venable et al., 2016). As some evaluation regarding the artifact is mentioned in chapter 3, some same points are described here that affect the whole thesis as a work of design science. The FEDS describes four different strategies for approaching the evaluation. One of the strategies is called Quick & Simple as it mainly focuses on two or just one evaluation at the end of the research process and it is often used in smaller research projects. As this thesis is a relatively small project that ends to a successful implementation the Quick & Simple strategy is a good fit for this thesis. Regardless of the strategy selected, they all are in a table that has two different axes from artificial to naturalistic and formative to summative. This thesis is clearly naturalistic because a real tool is implemented to a real product in real environment and comments and feedback is gathered from real users. This also shows the research rigor that has been present as comments from the real users from the company are used to define the results as a success and that the requirements are met, even if there are no long-term evidence of the success of this implementation project. Between formative and summative this thesis falls more to the summative evaluation as the evaluation happens to the ready implementation and results. This means that it is more about confirming that the set requirements are met. With these points Quick & Simple strategy is proven as a good strategy for this thesis and evaluated with it, but there are some possible risks.

The biggest risks considering this thesis come from research bias, because this is done by one person. However this risk is mitigated by means that are explained in (Romano et al., 2020) and are as follows. The fact that this thesis openly describes and explains the whole process, makes this less likely to have biased thoughts that are hidden, since every reader can make their own assumptions from the presented open information. Second part is that this is not a one-of-a-kind work, as the previously mentioned related works are highly similar in nature. In their results Romano found thoughts that if multiple different researchers publish confirmed experimental results, they are less likely to contain research bias (Romano et al., 2020). This however cannot be confused with opinions and comments from the company as this work is still done for them. There is also a possibility that during this thesis bias has influenced the company and therefore their comments, but that chance is small as described in the implementation process, options were shown openly and equally. Risks other than bias still exist but they are smaller compared to research bias. As selection process for tools and implementation phase is described openly, largest risk is possibly incomplete initial selection of testing tools for this thesis.

7 Conclusion

The goal of this thesis was to find and implement a GUI testing solution for the company. Tools were listed, filtered until there were few final candidates and then tested against each other. Selected testing tool was WebdriverIO as it fit to the company’s requirements and was better than other candidates in the company’s opinion. WebdriverIO was implemented to the given small one-page Vue.js application and to the company’s development pipeline. To achieve this, a mock-up for the API calls was created with Mock Service Worker. The end results are that GUI tests can be recorded with Chrome developer tools recorder or written, run them locally and in the company’s Jenkins server and the results are displayed in different reports for JUnit-report and HTML-report. These reports are integrated to Jenkins and are viewable from the Jenkins dashboard.

The testing tool WebdriverIO is the design science artifact in the form of instantiation as this thesis focuses on its implementation in addition to the tool itself. It is evaluated as a case study since its effects to the company environment are also considered. This is in the form of knowledge about GUI testing, testing automation, tools and practices that are researched and applied during the implementation process of WebdriverIO. The tool is mainly evaluated against the company’s requirements and following that other surrounding things are evaluated like the created mock-up.

Researched theory about testing revealed that there are places for both manual and automatic GUI testing and it is highly dependent about the target of the implementation. GUI testing should be approached in the beginning in the form of smoke tests that test basic functionality, because they are easier to make and show if there are any critical errors in base use. These smoke tests can then be adapted deeper and to more difficult features but doing so makes them more complex, easier to break and harder to maintain. This is a balanced line that the company must find themselves and what fits to its resources. When adapting to larger multiple page applications link and page coverage are good candidates for success criteria. With GUI testing the basic testing principles also apply, like using boundary value analysis and equivalence partitioning in input values or list of elements. Testing modern web applications brings dynamic content, which limits the use of more traditional testing methods and strategies.

The most important part still is that the software is made testable. In short this means visibility and control, like the program is structured so that it is easy to pick one feature and test only that and testers have access to all necessary information. The HTML elements should be given unique id's so that it is easier to interact with then while using a GUI testing tool. This way complex selectors can be avoided, and tests become more resilient to changes. Changes, successes, and failures should appear in the GUI elements so that the testing tool can see them as it can only see what a user can, which in this case means HTML-elements. Test scripts themselves should be as simple as possible and focus on one thing to ease maintenance and tests should be designed when building the GUI to avoid extra work in the automation phase. This is because GUI changes are easier to make in earlier phases of the development. Testing automation implementation should also only be done to a stable software in a regression testing sense, to avoid dealing with continuously breaking tests.

This thesis describes the implementation of WebdriverIO testing framework to a Vue.js based web application with a mock-up made with Mock Service Worker and the steps taken to ensure this whole package works in Jenkins that runs on a FreeBSD based server. It was limited to this single implementation on these platforms, technologies, and configurations as there can be uncountable number of possible ones. As there are no previous studies that have had this exact problem, now others can use this as a baseline to create their own implementations in the future. With the information provided in the appendices, an implementation can be made with little effort even if the testable software does not use Vue.js, Jenkins does not run-on FreeBSD, or the mock-up is created with some other way.

In the future this work could be expanded by creating multiple implementations to different web applications. Then test different testing methodologies and strategies and gather their advantages and disadvantages on a modern setting to be used on new implementations.

References

35+ Best GUI Testing Tools with Complete Details [2022 LIST] [WWW Document], n.d. . Softw. Test. Help. URL https://www.softwaretestinghelp.com/best-gui-testing-tools/ (accessed 6.7.22).

58 Best Automation Testing Tools: The Ultimate Guide | Test Guild [WWW Document], n.d. . Autom. Test. Made Easy Tools Tips Train. URL https://testguild.com/automation-testing-tools/ (accessed 6.7.22).

A. M. Memon, 2002. GUI testing: pitfalls and process. Computer 35, 87–88. https://doi.org/10.1109/MC.2002.1023795

Alégroth, E., Ardito, L., Coppola, R., Feldt, R., 2021. Special issue on new generations of UI testing. Softw. Test. Verification Reliab. 31, n/a. https://doi.org/10.1002/stvr.1770

Almenar, F., Esparcia-Alcázar, A.I., Martínez, M., Rueda, U., 2016. Automated Testing of Web Applications with TESTAR, in: Sarro, F., Deb, K. (Eds.), Search Based Software Engineering. Springer International Publishing, Cham, pp. 218–223.

Alturki, A., Bandara, W., Gable, G.G., 2012. Design Science Research and the Core of Information Systems, in: Peffers, K., Rothenberger, M., Kuechler, B. (Eds.), Design Science Research in Information Systems. Advances in Theory and Practice. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 309–327.

Banerjee, I., Nguyen, B., Garousi, V., Memon, A., 2013. Graphical user interface (GUI) testing: Systematic mapping and repository. Inf. Softw. Technol. 55, 1679–1694. https://doi.org/10.1016/j.infsof.2013.03.004

Blackburn, M., Busser, R., Nauman, A., 2004. Why model-based test automation is different and what you should know to get started, in: International Conference on Practical Software Quality and Testing. pp. 212–232.

BrowserStack blog [WWW Document], 2021. . Brows. Blog. URL https://www.browserstack.com/blog/browserstack-acquires-nightwatch/ (accessed 9.2.22).

Bruns, A., Kornstadt, A., Wichmann, D., 2009. Web Application Tests with Selenium. IEEE Softw. 26, 88–91. https://doi.org/10.1109/MS.2009.144

C. Persson, N. Yilmazturk, 2004. Establishment of automated regression testing at ABB: industrial experience report on “avoiding the pitfalls,” in: Proceedings. 19th International Conference on Automated Software Engineering, 2004. Presented at the Proceedings. 19th International Conference on Automated Software Engineering, 2004., pp. 112–121. https://doi.org/10.1109/ASE.2004.1342729

Carino, S., Andrews, J., 2015. Dynamically testing GUIs using ant colony optimization, in: ASE, ASE ’15. Presented at the Automated Software Engineering, IEEE Press, pp. 138–148. https://doi.org/10.1109/ASE.2015.70

Cleven, A., Gubler, P., Hüner, K., 2009. Design alternatives for the evaluation of design science research artifacts, in: DESRIST ’09. Presented at the Design Science Research In Information Systems And Technologies, ACM, pp. 1–8. https://doi.org/10.1145/1555619.1555645

Comparison of GUI testing tools - Wikipedia [WWW Document], n.d. URL https://en.wikipedia.org/wiki/Comparison_of_GUI_testing_tools (accessed 6.7.22).

Conroy, K.M., Grechanik, M., Hellige, M., Liongosari, E.S., Qing Xie, 2007. Automatic Test Generation From GUI Applications For Testing Web Services, in: ICSM. IEEE, pp. 345–354. https://doi.org/10.1109/ICSM.2007.4362647

CSP Reference & Examples [WWW Document], n.d. URL https://content-security-policy.com/ (accessed 9.8.22).

Di Lucca, G.A., Fasolino, A.R., 2006. Testing Web-based applications: The state of the art and future trends. Qual. Assur. Test. Web-Based Appl. 48, 1172–1186. https://doi.org/10.1016/j.infsof.2006.06.006

Ermuth, M., Pradel, M., 2016. Monkey see, monkey do: effective generation of GUI tests with inferred macro events, in: ISSTA 2016. Presented at the International Symposium on Software Testing and Analysis, ACM, pp. 82–93. https://doi.org/10.1145/2931037.2931053

García, B., Gallego, M., Gortázar, F., Munoz-Organero, M., 2020. A Survey of the Selenium Ecosystem. Electron. Basel 9, 1067. https://doi.org/10.3390/electronics9071067

Heinonen, J., 2020. Design and implementation of automated visual regression testing in a large software product. LUTPub

Hellmann, T.D., Hosseini-Khayat, A., Maurer, F., 2010. Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design, in: ICSTW. IEEE, pp. 444–447. https://doi.org/10.1109/ICSTW.2010.35

Hevner, A., Chatterjee, S., 2010. Design Science Research in Information Systems, in: Hevner, A., Chatterjee, S. (Eds.), Design Research in Information Systems: Theory and Practice. Springer US, Boston, MA, pp. 9–22. https://doi.org/10.1007/978-1-4419-5653-8_2

Hevner, Alan., Chatterjee, Samir., 2010. Design Research in Information Systems Theory and Practice, Integrated Series in Information Systems, 22. Springer US, Boston, MA. https://doi.org/10.1007/978-1-4419-5653-8

Hevner, A.R., March, S.T., Park, J., Ram, S., 2004. Design Science in Information Systems Research. MIS Q. 28, 75–105. https://doi.org/10.2307/25148625

Hynninen, T., Kasurinen, J., Knutas, A., Taipale, O., 2018. Software testing: Survey of the industry practices, in: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). Presented at the 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, Opatija, pp. 1449–1454. https://doi.org/10.23919/MIPRO.2018.8400261

IEEE, 2022. ISO/IEC/IEEE International Standard - Software and systems engineering --Software testing --Part 1:General concepts. ISOIECIEEE 29119-12022E 1–60. https://doi.org/10.1109/IEEESTD.2022.9698145

IEEE, 2021a. IEEE/ISO/IEC International Standard - Software and systems engineering--Software testing--Part 4: Test techniques. ISOIECIEEE 29119-42021E 1–148. https://doi.org/10.1109/IEEESTD.2021.9591574

IEEE, 2021b. ISO/IEC/IEEE International Standard - Software and systems engineering - Software testing -- Part 2: Test processes. ISOIECIEEE 29119-22021E 1–64. https://doi.org/10.1109/IEEESTD.2021.9591508

J. A. Whittaker, 2000. What is software testing? And why is it so hard? IEEE Softw. 17, 70–79. https://doi.org/10.1109/52.819971

Jenkins User Documentation [WWW Document], n.d. . Jenkins User Doc. URL https://www.jenkins.io/doc/ (accessed 9.8.22).

K. Karhu, T. Repo, O. Taipale, K. Smolander, 2009. Empirical Observations on Software Testing Automation, in: 2009 International Conference on Software Testing Verification and Validation. Presented at the 2009 International Conference on Software Testing Verification and Validation, pp. 201–209. https://doi.org/10.1109/ICST.2009.16

Kandil, M., Hassanein, E., Mazen, S., 2012. Cross-View of Testing Techniques Toward Improving Web-Based Application Testing. Int. J. Comput. Sci. Issues 9, 271–271.

Kasurinen, J., Taipale, O., Smolander, K., 2010. Software Test Automation in Practice: Empirical Observations. Adv. Softw. Eng. 2010, 1–18. https://doi.org/10.1155/2010/620836

Lahtinen, M., 2022. Implementing UI regression testing process to existing web application. LUTPub

Lee, J., Wyner, G.M., Pentland, B.T., 2008. Process Grammar as a Tool for Business Process Design. MIS Q. 32, 757–778. https://doi.org/10.2307/25148871

Mahmud, J., Cypher, A., Haber, E., Lau, T., 2014. Design and industrial evaluation of a tool supporting semi-automated website testing. Softw. Test. Verification Reliab. 24, 61–82. https://doi.org/10.1002/stvr.1484

Mattiello, G.R., Endo, A.T., 2021. Model-based testing leveraged for automated web tests. Softw. Qual. J. https://doi.org/10.1007/s11219-021-09575-w

Memon, A., Nagarajan, A., Xie, Q., 2005. Automating regression testing for evolving GUI software. J. Softw. Maint. Evol. Res. Pract. 12, 27–64. https://doi.org/10.1002/smr.305

Memon, A.M., 2007. An event-flow model of GUI-based applications for testing. Softw. Test. Verification Reliab. 17, 137–157. https://doi.org/10.1002/stvr.364

Memon, A.M., Nguyen, B.N., 2010. Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End, in: Zelkowitz, M.V. (Ed.), Advances in Computers. Elsevier, pp. 121–162. https://doi.org/10.1016/S0065-2458(10)80003-8

Memon, A.M., Pollack, M.E., Soffa, M.L., 2001. Hierarchical GUI test case generation using automated planning. IEEE Trans. Softw. Eng. 27, 144–155. https://doi.org/10.1109/32.908959

Mock Service Worker [WWW Document], n.d. . Mock Serv. Work. URL https://mswjs.io/ (accessed 9.7.22).

Molnar, A., 2012. An initial study on ideal GUI test case replayability, in: AQTR. IEEE, pp. 376–381. https://doi.org/10.1109/AQTR.2012.6237736

Moreira, R.M.L.M., Paiva, A.C., Nabuco, M., Memon, A., 2017. Pattern-based GUI testing: Bridging the gap between design and quality assurance. Softw. Test. Verification Reliab. 27, np-n/a. https://doi.org/10.1002/stvr.1629

Moreira, R.M.L.M., Paiva, A.C.R., 2014. A GUI modeling DSL for pattern-based GUI testing PARADIGM, in: ENASE. SCITEPRESS, pp. 1–10.

Myers, G.J., Badgett, Tom., Thomas, T.M., Sandler, Corey., 2004. The art of software testing, 2nd ed. ed. Wiley, Hoboken (NJ).

Myers, G.J., Sandler, C., Badgett, T., 2012. The art of software testing, 3rd ed. ed. John Wiley & Sons, Hoboken, N.J.

Nunamaker, J.F., Chen, M., Purdin, T.D., 1990. Systems Development in Information Systems Research. J. Manag. Inf. Syst. 7, 89–106. https://doi.org/10.1080/07421222.1990.11517898

Okezie, F., Odun-Ayo, I., Bogle, S., 2019. A Critical Analysis of Software Testing Tools. J. Phys. Conf. Ser. 1378, 42030. https://doi.org/10.1088/1742-6596/1378/4/042030

Peffers, K., Rothenberger, M., Tuunanen, T., Vaezi, R., 2012. Design Science Research Evaluation, in: Peffers, K., Rothenberger, M., Kuechler, B. (Eds.), Design Science Research in Information Systems. Advances in Theory and Practice. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 398–410.

Peffers, K., Tuunanen, T., Gengler, C.E., Rossi, M., Hui, W., Virtanen, V., Bragge, J., 2006. The design science research process: A model for producing and presenting information systems research, in: Proceedings of the First International Conference on Design Science Research in Information Systems and Technology (DESRIST 2006), Claremont, CA, USA. pp. 83–106.

Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S., 2007. A Design Science Research Methodology for Information Systems Research. J. Manag. Inf. Syst. 24, 45–77. https://doi.org/10.2753/MIS0742-1222240302

Peffers, Ken., 2012. Design science research in information systems : advances in theory and practice : 7th international conference, DESRIST 2012, Las Vegas, NV, USA, May 14-15, 2012. Proceedings / Editors: Ken Peffers, Marcus Rothenberger, Bill Kuechler., Lecture notes in computer science, 7286. Springer, Berlin.

Pettichord, B., Bach, J., Kaner, C., 2011. Lessons Learned in Software Testing. Wiley-Blackwell.

Pöysä, S., 2019. Integrating unit testing into an organization with legacy ASP.NET applications. LUTPub

Romano, S., Fucci, D., Scanniello, G., Baldassarre, M.T., Turhan, B., Juristo, N., 2020. Researcher Bias in Software Engineering Experiments: a Qualitative Investigation. arXiv.org. https://doi.org/10.48550/arXiv.2008.12528

S. Berner, R. Weber, R. K. Keller, 2005. Observations and lessons learned from automated testing, in: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. Presented at the Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., pp. 571–579. https://doi.org/10.1109/ICSE.2005.1553603

Sabev, P., Grigorova, K., 2017. A Comparative Study of GUI Automated Tools for Software Testing, in: Proc. of The Third International Conference on Advances and Trends in Software Engineering.

Salohonka, M., 2020. Automated testing of React Native applications. LUTPub

Salvatore T. March, Veda C. Storey, 2008. Design Science in the Information Systems Discipline: An Introduction to the Special Issue on Design Science Research. MIS Q. 32, 725–730. https://doi.org/10.2307/25148869

Santos-Neto, P., Resende, R., Pádua, C., 2007. Requirements for Information Systems Model-Based Testing, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC ’07. Association for Computing Machinery, New York, NY, USA, pp. 1409–1415. https://doi.org/10.1145/1244002.1244306

Silistre, A., Kilincceker, O., Belli, F., Challenger, M., Kardas, G., 2020. Models in Graphical User Interface Testing: Study Design, in: UYMS. IEEE, pp. 1–6. https://doi.org/10.1109/UYMS50627.2020.9247072

Simon Bisson, 2020. Testing web applications with node.js and playwright. InfoWorld.com.

Singhera, Z., Horowitz, E., Shah, A., 2008. A Graphical User Interface (GUI) Testing Methodology. Int. J. Inf. Technol. Web Eng. 3, 1–18. https://doi.org/10.4018/jitwe.2008040101

Takeda, H., Veerkamp, P., Yoshikawa, H., 1990. Modeling Design Process. AI Mag. 11, 37. https://doi.org/10.1609/aimag.v11i4.855

Testing | Vue.js [WWW Document], n.d. URL https://vuejs.org/ (accessed 7.1.22).

Venable, J., Pries-Heje, J., Baskerville, R., 2016. FEDS: a Framework for Evaluation in Design Science Research. Eur. J. Inf. Syst. 25, 77–89. https://doi.org/10.1057/ejis.2014.36

Venable, J., Pries-Heje, J., Baskerville, R., 2012. A Comprehensive Framework for Evaluation in Design Science Research, in: Peffers, K., Rothenberger, M., Kuechler, B. (Eds.), Design Science Research in Information Systems. Advances in Theory and Practice. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 423–438.

Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., Kazmeier, J., 2006. Automation of GUI Testing Using a Model-Driven Approach, in: Proceedings of the 2006 International Workshop on Automation of Software Test, AST '06. Association for Computing Machinery, New York, NY, USA, pp. 9–14. https://doi.org/10.1145/1138929.1138932

WebdriverIO [WWW Document], n.d. URL https://webdriver.io/docs/gettingstarted (accessed 9.8.22).

Winter, R., 2008. Design science research in Europe. Eur. J. Inf. Syst. 17, 470–475. https://doi.org/10.1057/ejis.2008.44

X. Zhu, B. Zhou, J. Li, Q. Gao, 2008. A test automation solution on GUI functional test, in: 2008 6th IEEE International Conference on Industrial Informatics. Presented at the 2008 6th IEEE International Conference on Industrial Informatics, pp. 1413–1418. https://doi.org/10.1109/INDIN.2008.4618325

Zhyhulin, D., Kasian, K., Kasian, M., 2022. Combined method of prioritization and automation of software regression testing, in: TCSET. IEEE, Piscataway, pp. 751–755. https://doi.org/10.1109/TCSET55632.2022.9767034

Appendix 1. Testing tools

OS: W – Windows, L – Linux, M – MacOS

Update status (ST): U - Updated, D – Deprecated, K - Dead

Name	Vendor	URL	Latest version	OS	Supported technologies	License	Demo	ST	Last update
Abbot Java GUI Test Framework	open source	https://sourceforge.net/projects/abbot/	1.2.0	WL M	java	common public 1.0		D	12/2011
Appium	open source	https://appium.io	1.22.3	WL M	java, c#, javascript, php,python, ruby, robot framework	apache-2.0		U	03/2022
Ascential-Test	Zeenyx	https://zeenyx.com	AT 9.8.0	W	Java, Js, Ajax, AngularJS, HTML5, REST	proprietary	30d trial on request	U	12/2021
AutoIt	AutoIt	https://www.autoitscript.com/site/	3.3.16.0	W				U	03/2022
Coded UI	microsoft	https://docs.microsoft.com/en-us/visualstudio/test/?view=vs-2022		W		proprietary		D	
CubicTest	open source	https://github.com/cubictest/cubictest		W	java	eclipse public license 1.0		D	05/2012
CucumberStudio	smartbear	https://cucumber.io/	8.2.1		most mainstream languages	proprietary	14d trial	U	05/2022
Cypress	open source	https://www.cypress.io/	9.6.1	WL M	Javascript	MIT		U	05/2022
eggPlant UI Automation Testing	keysight	https://www.eggplantsoftware.com/	22.1	WL M	any	proprietary	on request	U	04/2022
FitNesse	open source	http://fitnesse.org/FrontPage	20220319		java, others with plugins	Common public license 1.0		U	03/2022
Gauge + Taiko	open source	https://gauge.org/	1.4.3	WL M	Javascript, c#, Java, Python, Ruby, markdown	apache-2.0		U	01/2022

GTT	open source	https://sourceforge.net/projects/gtt/	3.0	W	java	GNU General public license 2.0		D	11/2009
GUITAR	open source	https://sourceforge.net/projects/guitar/				GNU General public license 3.0		D	2012
IcuTest								K	
iMacros	progress	https://www.progress.com/imacros	14.2.4.18	W	most mainstream languages	proprietary	30d trial	U	02/2022
Jubula GUI testing tool	open source	https://www.bredex.de/jubula	8.8.1	WLM	java/code free	eclipse public license		U	01/2022
Katalon studio	katalon	https://katalon.com/web-testing	8.3	WLM	HTML5, javascript, Ajax, Angular	proprietary	free version with less features	U	04/2022
Maveryx user interface testing tool	maveryx	https://www.maveryx.com/	2.5.0	WLM	HTML5, javascript, Angular, Node.js, React, Electron…	proprietary	on request	U	11/2021
Micro Focus Unified Functional Testing (UFT)	microfocus	https://www.microfocus.com/en-us/products/uft-one/overview	2021 R1	W	java, 200+ different	proprietary	30d trial on request	U	2021
Nightwatch	open source	https://nightwatchjs.org/	2.1.6	WLM	Javascript, typescript	MIT		U	05/2022
OpenTest	open source	https://getopentest.org/	1.2.4	WLM	Javascript	MIT		U	09/2020
Oxygen	open source	https://oxygenhq.org/	1.27.0	WLM	Javascript	GNU General public license 3.0		U	03/2022
Parasoft Selenic	parasoft	https://www.parasoft.com/products/parasoft-selenic/	2021.1	WLM	Java	proprietary	on request	U	08/2021
Playwright	open source	https://playwright.dev/	1.22.1	WLM	Javascript, Typescript, Java, Python, .NET	apache-2.0		U	05/2022

Pyle-nium.io	open source	https://docs.pyle-nium.io/	1.15.2	WLM	Python	MIT		U	04/2022
Qaliber	open source	https://sour-ceforge.net/pro-jects/qaliber/		W				D	03/2011
QF-Test	qfs	https://www.qfs.de/en/index.html	5.4.3	WLM	java apps, HTML5, Ajax, native win apps, an-droid, angu-lar, vue, vaa-din, react	pro-prietary	30d trial on request	U	03/2022
Ranorex Studio	ranorex	https://www.rano-rex.com/	10.1.7	WLM	c#, VB.NET	pro-prietary	on re-quest	U	03/2022
Rapise	inflectra	https://www.in-flectra.com/Rapise/	7.2	W	html5, ajax, java apps, ios, android, win apps	pro-prietary	30d trial	U	02/2022
RCP Tes-ting Tool	open source	https://www.ec-lipse.org/rcptt/	2.5.3	WLM	java	eclipse public li-cense 1.0		U	05/2022
ReadyAPI	smart-bear	https://smart-bear.com/pro-duct/ready-api/over-view/	3.30.0	W		pro-prietary		U	03/2022
RIATest								K	
Robot Fra-mework	open source	https://robotfra-mework.org/	5.0.1	WLM	Python	apache-2.0		U	05/2022
Sahi	sahipro	https://www.sa-hipro.com/	9.7.0	WLM	javascript, java,	pro-prietary	on re-quest	U	
Selenide	open source	https://selenide.org/	6.5.0	WLM	java	MIT		U	05/2022
Selenium	open source	https://www.sele-nium.dev/	4.1.0	WLM	java, python, c#, ruby, ja-vascript, kot-lin	apache-2.0		U	11/2021
Selenium-Base	open source	https://selenium-base.com/	3.1.1	WLM	Python	MIT		U	05/2022
Serenity	open source	https://serenity-bdd.info/	3.2.0	WLM	Java	apache-2.0		U	02/2022
Sikuli UI Automation framework								K	

SilkTest	mic-rofocus	https://www.microfo-cus.com/en-us/pro-ducts/silk-test/over-view	21.0.1	WLM	ajax, html5, ios, android, java apps, win apps	pro-prietary	30d trial	U	09/2021
Squish GUI Testing tool	froglo-gic	https://www.froglo-gic.com/squish/	7.0	WLM	qt, java, tk, vnc, html5,	pro-prietary	on re-quest	U	03/2022
SWTBot	open source	https://www.ec-lipse.org/swtbot/	3.1.0	WLM	Java	eclipse public li-cense 2.0		U	06/2021
Telerik Testing Framework	telerik	https://www.tele-rik.com/teststu-dio/testing-fra-mework	R1 2022 SP2(v. 2022.1.6 01)		.net, html5 ja-vascript, wpf	pro-prietary		U	05/2022
Tellurium Automated Testing Framework	open source	https://code.google.co m/archive/p/aost/				apache-2.0		D	
Test Studio	telerik	https://www.tele-rik.com/teststudio	R1 2022 SP1	W	c#, VB.NET	pro-prietary	on re-quest	U	04/2022
TestComp-lete	smart-bear	https://smart-bear.com/pro-duct/testcomplete	15.40	W	.net, c++, del-phi, java, an-gularJS, qt, react	pro-prietary	on re-quest	U	03/2022
TestIM	testim a tricentis com-pany	https://www.tes-tim.io/test-automa-tion-tool/	v2.0	W	Javascript	pro-prietary	on re-quest	U	03/2022
TestIQ	Au-tonomI Q	https://au-tonomiq.io/in-dex.html				pro-prietary	on re-quest	U	
TestPartner								K	
TestProject	open source	https://testproject.io/	3.5.0	WLM	Javascript, Java, Python, c#	pro-prietary	on re-quest	U	04/2022
TestS-tack.White Framework	open source	https://www.nuget.or g/packages/TestS-tack.White	0.13.3	W		apache-2.0		D	12/2014
Twist								K	
UI Auto-mation Po-wershell Extensions	open source	https://www.open-hub.net/p/UIA		W		GNU Ge-neral public li-cense 2.0		D	2016
Watir	open source	http://watir.com/	7.1	WLM	ruby	MIT		U	09/2021
Webdrive-rIO	open source	https://webdriver.io/	7.19.5	WLM	Javascript	MIT		U	05/2022

Appendix 2. Used commands

install wdio	npm init wdio
run all tests with wdio	npm run wdio
install chromedriver service	npm install wdio-chromedriver-service --save-dev
install chromedriver	npm install chromedriver --save-dev
install firefox driver service	npm install wdio-geckodriver-service --save-dev
install firefox driver	npm install geckodriver --save-dev
install mock service worker	npm install msw --save-dev
generate worker for mock service worker	npx msw init public/ --save
install JUnit reporter	npm install @wdio/junit-reporter --save-dev
install html reporter	npm install wdio-html-nice-reporter
install video reporter	npm install wdio-video-reporter
install chrome recorder translator	npm install @wdio/chrome-recorder
run the record translator	npx @wdio/chrome recorder
run with a specific configuration file	npx wdio run **CONFIGURATION-FILE** ex: npx wdio run wdio.conf.js
run a single test	npx wdio run **CONFIGURATION-FILE** --spec **PATH-TO-TEST-FILE** ex: npx wdio run wdio.conf.js --spec test/specs/example.js
Jenkins content security policy setting	System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "sandbox allow-scripts; default-src 'self'; img-src 'self' data:; style-src 'unsafe-inline'; script-src 'unsafe-inline'; style-src-elem 'self'; script-src-elem 'unsafe-inline' 'self';")

Appendix 3. Jenkins file

```
pipeline {
    agent any
    tools {
        gradle "gradle-6.2.2"
        jdk "openjdk11"
    }
    stages {
        stage('build') {
            steps {
                checkout scm
                sh "gradle clean build --stacktrace"
            }
        }
        stage('test'){
            parallel {
                stage('Run dev server'){
                    steps {
                        timeout(unit: 'MINUTES',time:2){
                            script {
                                try {
                                    sh "npm run serve"
                                }
                                catch (e){
                                    if (e ==
org.jenkinsci.plugins.workflow.steps.FlowInterruptedException){
                                        currentBuild.result = 'SUCCESS'
                                        return
                                    }}}}}}
                stage('Run tests'){
                    steps {
                        sleep(20)
                        sh "npm run wdio"
                    }
                    post {
                        always {
                            junit "test_output/*.xml"
                            publishHTML (target : [allowMissing: false,
                            alwaysLinkToLastBuild: true,
                            keepAll: true,
                            reportDir: 'reports/html-reports/',
                            reportFiles: 'master-report.html',
                            reportName: 'forecast wdio html test report'])
                        }}}}}}}
```

Appendix 4. WebdriverIO configuration file changes

Otherwise, the configuration file is in its default settings except for these changes.

At the top of the file for the HTML reporter before export.

```
const {ReportAggregator, HtmlReporter} = require('wdio-html-nice-reporter');
```

Capabilities for the chrome browser

```
    capabilities: [
    {
        browserName: 'chrome',
        acceptInsecureCerts: true,
        "goog:chromeOptions": {
            args:[
                '--headless'
            ]
        }
    }
    ],
```

Services installed, has the chromedriver service. For Jenkins configuration file there is also a chromedriverCustomPath: argument that was used to make sure that right driver is used.

```
    services: [
        [
            'chromedriver',
           {
            logFileName: 'wdio-chromedriver.log',
            outputDir: './logs/chrome',
           }]
    ],
```

Reporters installed, has HTML, JUnit, and the default spec-reporter.

```
    reporters: ['spec',
        ['junit', {
            outputDir: 'test_output',
            outputFileFormat: function(options) {
                return `results-${options.cid}.${options.capabilities}.xml`
            }
        }],
        ["html-nice", {
            outputDir: 'reports/html-reports',
            filename: 'report.html',
            reportTitle: 'Test Report Title',
            linkScreenshots: true,
            showInBrowser: false,
            collapseTests: false,
            useOnAfterCommandForScreenshot: true
        }
        ]
    ],
```

onPrepare hook for the HTML reporter.

```
    onPrepare: function (config, capabilities) {
        reportAggregator = new ReportAggregator({
            outputDir: 'reports/html-reports',
            filename: 'master-report.html',
            reportTitle: 'Master Report',
            browserName : capabilities.browserName,
            collapseTests: true
          });
        reportAggregator.clean();
    },
```

afterTest hook for taking screenshot on failure

```
afterTest: function(test, context, {error, result, duration, passed, retries})
{
    const date = (new Date()).toString().split(` `,6);
    date.pop();
    date.shift();
    const dateString = (date.join('_')).replace(new RegExp(`:`, `g`), ``);
    if (error){
            browser.saveScreenshot('reports/html-
            reportsscreenshots/'+dateString+'_screenshot.png');
    }
},
```

onComplete hook for creating the html report

```
onComplete: function(exitCode, config, capabilities, results) {
    (async () => {
        await reportAggregator.createReport();
    })();
},
```