Testing React Single Page Web Application using Automated Testing Tools

Md Mehedee Hasan[1], Mohammad Ashikur Rahman[1], Md Salman Chowdhury[1], Md Habibur Rahman[1], Kaal Harir Abdulle[1], Farzana Sadia[2] and Mahady Hasan[1]
[1]*Department of Computer Science and Engineering, Independent University, Bangladesh, Dhaka, Bangladesh*
[2]*Department of Software Engineering, Daffodil International University, Dhaka, Bangladesh*

Keywords: Software Testing, Testing Tools, GUI Testing, Automated Testing Tools, Single Page Web Application Testing, React Application Testing.

Abstract: In modern software development practices, single page application development is becoming popular. Among the available frameworks, React is a popular platform to develop web applications front-end. For most of the software, almost half of the application code is for the front-end. Therefore testing the front-end or GUI is also almost equally important. In modern application development, React is by far the most popular front-end framework/library. In modern development practice, like agile development or CI/CD development, testing performance and the easiness of implementing testing tools are very crucial factor. In this paper, three different testing tools, Jest, Enzyme and Cypress specifically designed for React based single page application, were analysed. The main objective is to check the execution time performance for the same testing functionality on different tools, and developer friendliness of these tools. While each tool having their own features and limitations, for modern development practices like Agile or CI/CD, Enzyme is tested as a better tool which is both easy to integrate while providing efficiency.

1 INTRODUCTION

In modern web development, testing has become a very important part in software development. Ensuring the quality of the product is also becoming important. While testing the business logic at the application level ensures the reliability of the applications functionality, it is also important to test the Graphical User Interface (GUI) to provide users a more seamless experience. In modern software applications, around 50% of the application code is for GUI implementation. (Broer Bahaweres et al., 2020). The modern development can be divided into two segments, namely, a) frontend development, b) backend development. Front-end web development is the development of the graphical user interface of a website, through the use of HTML, CSS, and JavaScript, so that users can view and interact with that website.(b15, 2021). React is one of the most popular and widely used libraries on modern websites. This library can be used for either single page application (SPA) or even mobile applications (React Native). Testing the applications based on React can be slightly different during development. Generic testing techniques might not be effective for such library based GUI (Xiao-Fang Qi, 2017)

When testing GUI, we can have two approaches, a) Manual Testing where one or more person is manually testing the application and b) Automated testing where the tests are done based on a written test code automatically. Manual testing has some advantages, however, it requires a dedicated QA team (Chen et al., 2020), manual testing is often costly (Broer Bahaweres et al., 2020) (Garousi and Yildirim, 2018), and manual testing can be error prone due to human interaction (Silistre et al., 2020). In case of automated testing, it does require programming knowledge, but it provides a lot of benefits over manual testing. In industrial level touch based machines automated testing provides several advantages(Klammer and Ramler, 2017a), it is also beneficial for modern development practice like Continuous Integration and Continuous Deployment (CI/CD) (Alégroth et al., 2018).

While recent studies mostly focuses on particular tools for testing, there is a little study on how different tools performs and how easy to work with them with Single Page applications, like applications built on React. To find out how different tools can affect

Hasan, M., Rahman, M., Chowdhury, M., Rahman, M., Abdulle, K., Sadia, F. and Hasan, M.
Testing React Single Page Web Application using Automated Testing Tools.
DOI: 10.5220/0011077900003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 469-476
ISBN: 978-989-758-568-5; ISSN: 2184-4895

the single page applications we developed three main research objects, a) Easiness of integration of different tools, b) How different tool performs for similar tests and C) Developer friendliness to write test codes using different tools. In this paper, we will try to study three different testing tools named, Jest, Enzyme and Cypress and explore their various attributes like performance, user friendliness, memory cost on React based websites. Along with our study, we will further find the benefits and drawbacks of these tools to propose a suitable tool for different levels of software development which in return, can provide better software quality with interactive GUI.

2 LITERATURE REVIEW

Modern applications are separated into frontend and backend development. Backend development is responsible for the functionality of the business cases and processing data. One of the benefits of separating two parts is, the frontend or the backend can be changed without changing or with very little changes on the other end. This often provides the developers to change the backend language without changing the User experience in the frontend. The frontend is responsible for handling the user interface through Graphical User Interface (GUI) (b15, 2021). The frontend is often developed as a single page application (SPA) to provide users a more immersive and interactive experience. Unlike the traditional web application development where each interaction is achieved through different pages using HTML and CSS, such interactive pages often take benefits of javascript, taking the benefits of providing users the similar experience without refreshing the pages. The concept of (Asynchronous JavaScript and XML) AJAX is hugely implemented here. There are various libraries and frameworks used to develop interactive web interfaces. In recent years, React is by far the most popular front-end framework/library (and continues to grow faster) (2021,). One of the features that makes React stand out from traditional web development is its Virtual DOM functionality. The virtual dom provides benefits where the codebase is large. In regular DOM (Data Object Model), these are in simple words the html and stylesheets, however, if the project is too large, there can be thousands of lines of codes which makes it inconvenient and hard to manage. In React, these large codebases are divided into small components, which can be returned on user request or based on the data from the server side.

When testing the frontend or GUI both manual testing and automated testing is used. Manual testing provides some benefits but it is often costly and time consuming (Broer Bahaweres et al., 2020). Previous research by Filippo Cacciotto, Tommaso Fulcini, Riccardo Coppola, and Luca Ardito finds that manual testing can be boring, costly and repetitive tasks which are handled manually (Klammer and Ramler, 2017b). The author proposed gamification in manual testing which might convince the developers in manual testing. Another drawback of manual testing is that the test team needs to manually update the test scripts as the project becomes large (Aho et al., 2021). Automated testing is becoming more popular day by day due to the nature of modern software development practices. One of the most popular development practices, Agile development often follows Testing focused processes such as Test Driven Development (TDD), Behavior Driven Development (BDD). These practices are considered to play a very important role in the success of the project (Broer Bahaweres et al., 2020). Automated testing is also essential in Continuous Integration and Continuous Deployment (CI/CD) to make it work efficiently.(Alégroth et al., 2018). Automated testing tools like Selenium usually requires less maintenance but it requires some programming knowledge (Dobslaw et al., 2019). Muneyoshi Iyama even proposed a method to generate the automated test scripts, automatically based on static analysis and dynamic analysis on the application (Iyama et al., 2018). Automated testing tools also has the capability of taking benefit of modern technologies like Computer vision and Machine learning (Macchi et al., 2021) (White et al., 2019)

Most of the research focused on only improving the automated testing methodology or techniques means how efficiently the test cases can be executed with the different methodologies such as using machine learning or AI rather than focused on the testing tools and their efficiency. However, automated testing tools nowadays outweigh manual tools due to their many bright aspects. There exists a quite large number of GUI-testing tools all over the software industry and have been tremendously used as well. The goal is the same as with any testing technique - to identify a manageability small set of test cases that is sufficiently rigorous and diverse to expose any faults. However, concentrating on chiefly testing tools, especially automated testing tools in terms of their performance, for example, execution time, error detection rate, user friendly and so on are the key parameters or attributes in case of selecting testing tools. Many test engineers have challenges in the proper and successful implementation of test automation, especially in the case of VGT (Visual GUI Testing). Though some existing studies compare VGT tools, however, testing especially React components automatically that are based on java scripts are not much available in the industry for the testers. This paper aims to make a better understanding of the capabilities of these three tools and to assist the testers to evaluate the pros and cons associated with them.

VGT [Visual GUI Testing] tools require a specific expected output image to assert if what is being rendered is correct according to an expected result. The main implication of the findings provided is that VGT can be utilized in industrial practice for a long time (Alégroth and Feldt, 2017). However, many practicing test engineers have challenges in proper and successful implementation of test automation, especially in the case of VGT. when VGT and GUI test automation are not planned, designed, or implemented properly by test engineers, the efforts have led to disappointments and various negative outcomes (Garousi et al., 2017). Additionally, a new pattern-based GUI testing (PBGT) technique is introduced that formalizes the concept of UI Test Patterns, which are general test strategies for evaluating UI patterns across the many implementations and the findings show that PBGT is more effective than a manual model-based test case generation approach (Moreira et al., 2017).

The widespread use of iterative and incremental processes and continuous integration practices in software development has shortened the development cycles, drastically limiting the time for testing and quality assurance of each release. Instead of months or weeks, the longest period for testing a release is over a weekend or a night. The results of an automated smoke test set are expected almost instantly or in a few minutes. In practice, a high level of test automation is a requirement for a successful continuous integration process (Aho and Vos, 2018). However, evaluate the effectiveness of the characteristics on testing cost and fault detection is effectiveness and the test designer should take a diversity of states in which an event executes and the event coverage of the suite to get proper fault detection with effectiveness (Xie and Memon, 2006). Moreover, On-device test generation gives faster result for simple application which needs less computation and On-device test generation is 3.2 times faster than computer device testing (Borges et al., 2020).

React framework, unlike regular website, is a single page application. In regular websites, there are usually multiple pages containing different contents. React on the other hand, is a single page application, meaning all the contents will render on single page. React achieves this interactivity by rendering only the part requested by the user. Such parts are defined as Component in React. Single page applications are mostly Ajax based application which is heavily dependent on JavaScript language. It also uses special templating engine called JSX to render the html contents. Another special feature of single page application is Virtual DOM, where the HTML elements are dynamically updated or rendered. These uniqueness makes React applications different from regular websites, which also requires the testing tool to be precise in testing these components or virtually rendered dom elements or attributes.

In SPAs, all the required components are downloaded in the first page load. The twitter's timeline and the gmail's inbox are examples of SPA's. However, some contents and components are loaded dynamically as the user interacts with it. For example, in the same web page a user may change tabs open a dialog to post new content and execute pagination operations without changing the URL path in the browser's address bar. In a case where a web page generates UI components dynamically, for example a list of items appears as a consequence of a page change, or new items are loaded as the user scrolls, problems in the user experience (UX) may be perceived. These tools would not be able to identify it as something to be carefully analyzed by the developers because the scenario that causes the issue does not happen in the first page load. It depends on how the user interacts with the system and when the content is dynamically loaded (Quental et al., 2019). With the growing variety of screen sizes and resolutions of mobile devices, the need to validate GUI layout correctness in different screen sizes as well (Hasselknippe and Li, 2017).

React applications needs to be compiled before using, which can be time consuming based on the complexity and size of the application. However, in modern web development, time is one critical metric, on which the success of project highly depends. Therefore, the objective is to reduce the overall time taken to implement a feature and publish it with proper quality. And to ensure the quality, testing must be done before it is deployed.

Software development practices like Agile development or deployment scenarios like CI/CD are a continuous process where the features and functionalities are implemented over time in phases. In each iteration, the application also needs to pass through the testing phase as well. Study shows that, in automated testing, each attempt at a test case execution can take a long time, eliminating the possibility of rapidly attempting large numbers of alternatives (Walkinshaw, 2020). Our goal in this paper is to find a suitable tool which can be easily used to test the application GUI properly while it takes less time which improves overall application delivery time.

While the papers related to our field of study mostly focus on choosing manual or automated testing on GUI testing, there is little exploration on how different testing tools interact with such SPAs. Our goal in this project is to explore some of the modern automated testing tools for testing React based single page applications and observe different metrics which can provide insightful information.

2.1 Cypress

Cypress is a javascript library used for end-to-end testing. It can be used to test anything that runs on a browser. It is often compared with Selenium, but it has a totally different architecture. It facilitates different testings like End-to-end tests, Integration tests, Unit tests etc. Cypress is an open source software. It has a dashboard service which can record our test runs and produce logs. Setting up cypress is very easy. No servers, drivers or dependencies to install or configure. As an application is developed, the tests can be seen running in real time.

2.2 Enzyme

Enzyme is a JavaScript Testing utility for React that makes it easier to test your React Components' output. In this framework testers can manipulate and traverse the component. Learning curve is slightly Tough.

2.3 Jest

Jest is another popular testing tool for javascript. Jest is developed and maintained by Facebook Inc and widely popular among javascript developers. Jest can be used to test different types of projects both in backend and frontend. It focuses on simplicity to provide developers with faster testing. Jest is well documented and often recommended for React projects, however, it is not suitable for a beginner level developer.

3 METHODOLOGY

A lot of testing tools exist that are based on different architectural designs and implementations. As a consequence, it is not possible for researchers, and practitioners alike, to make an informed decision on which tool to use and on the strengths and weaknesses of the tools. Because of the nature of modern software development techniques, automated testing is becoming increasingly prevalent. In our research, we addressed this issue by comparing three automated testing tools that work for React projects. React has become a very popular and extensively used tool among front-end developers. The tools are Jest, Cypress and Enzyme. They are open source testing tools which are architecturally very different from each other. This paper compares the execution time, error detection rate, CPU usage and the ease of effort to set up or develop the tools.

To select our test projects, we performed a search on Github to choose an interactive React application on which we would implement the test scripts. We selected a popular React based app named TakeNote which has over 5100 stars and forked over 740 times. This is a simple note taking application which has the ability to create, edit, delete and update notes and can create categories. TakeNote, an interactive app, will be ideal for putting the three tools to the test. The data from the tools will be stored and compared for a full review of the tools. The testers are divided into three groups and each group assigned to each of the three tools. A list of test cases are prepared and the groups implement all the test cases with their respective tools. After running the test cases, the data is collected and compared. The running time, time to set up, ease of integration, error detection rates would give us enough data to analyse and provide insights for software business ranging from startups to enterprises.

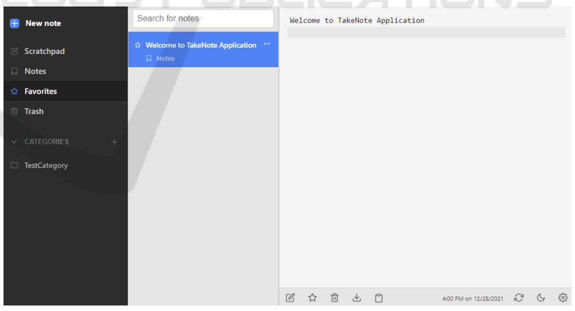

Figure 1: GUI of TakeNote Application.

Writing test cases can be error-prone and a tester's ability or skill level to write a test script may produce a bias for different tools. And also, the very different way cypress behaves from the other two testing tools may provide a challenge in reaching a solid conclusion. In order to minimize these issues, the test scripts are shared between the testers to prevent the scripts from being too different in terms of code complexity. Another limitation of our experiment that the number of tests are relatively small and the project is comparatively simpler. The real world application may have other dependencies which might force the developer team to go with specific tool regardless the benefit or limitation of any tool.

4 RESULT ANALYSIS

We targeted to respond to the following main research questions (RQs) in our study: How do these three tools compare when they are used to test React components from a single web page running the same test scripts? To evaluate the RQs methodically and for various features, we separated them into three sub-research questions:

- RQ1: How do the three tools compare in terms of integration?
- RQ2: How do the three tools compare in terms of "execution time" for the same test case?
- RQ3: How do the three tools compare in terms of user friendliness?

Before the tests were actually ran, we needed to integrate these tools with the project. We forked the main application from GitHub and created three different branch for each tools. And then we implemented our tools with configuration and packages. During our implementations, we came across some of the differences among these tools. In modern React applications, the Jest is provided as default testing tools with the default project skeleton. Both Enzyme and Cypress needs to be added separately. Jest while itself can not interact with the GUI directly, it depends on JEST-DOM and testing-packages to interact with the GUI elements. Enzyme operates similarly. However the main benefit of Jest is that it can be used to test the internal functionalities of React Components. Documentation from Jest shows that it can be used to test in business logic & functionalities as well. Enzyme performs similarly, however, it has the benefit to test a component without the need to render other components within that component. Cypress, on the other hand focuses on GUI. It has a native GUI where the tests can be simulated and monitored. However, if a developer is working on Linux system, cypress requires few operating system dependencies to be resolved before it can be operated. Once we have configured each tool with the project, we found the Jest to be easiest solution to start since it's already integrated, being Cypress the hardest one to configure since it has operating system library dependencies.

To observe how easy or hard to use each of these tool we explored different functionalities and facilities provided by each tools. Different GUI actions like Clicking, Typing, double click or navigating is not directly possible with Jest and Enzyme, rather relies on fireEvent, render or screen functionalities provided by testing-library/React package. These tools provides jest to trigger different events on GUI. En-

Table 1: Test Cases.

S/L	Test Scenario
1	User can click on the + button on Category menu and new category input field should appear, upon typing the name, when clicked outside, a new category should be created
2	Category list should be hidden when user clicks Category menu, if there are categories
3	When user clicks Sync button, a current time should be displayed as synced time.
5	When user visits the website, the application should be properly rendered
6	User can change to dark mode by clicking the 'Theme' button at bottom right
7	User can start a new note by pressing 'CTRL+ALT+N'
8	When a user clicks on a New note, it should take the user with a empty note
9	Settings modal should open when a user clicks on the settings icon
10	When user clicks the Scratchpad menu, it should take to the Scratchpad view
11	When user clicks on New note menu, it should open a new note
12	On the menu list a search box should be rendered which can be used to search notes
13	When the main view is loaded, Add new category menu should be rendered and ready for action
14	When user clicks on Preference tab in Settings modal, it should render the preference options
15	When user clicks on Data management tab in Settings modal, it should render the data management options
16	When user is on the data management tab, user should see an Export Button to export the notes
17	When user is on the data management tab, user should see an Import Button to import notes from other sources
18	When user is on the data management tab, user should see an Download all button to download
19	When user is on the About tab, user should see an View source Button to see the details about the app
20	User should see logout button when they are in Settings modal

zyme, in our observation performs similarly, since it depends on Jest by default, however this can be configured with other tool as well. Cypress on their other hand, provides a handful of features to interact with the different GUI elements. We found cypress very easy for writing the test codes, while both Jest and Enzyme has some learning curve.

To study the behavior of our selected tools, a number of tests were generated to test different elements. Then these tests were written using three tools, The dependencies, limitations and benefits of integrating and writing the test codes were closely logged during developing the test codes. Table 1 contains the test cases defined for our study

After test codes were written and prepared using three different tools, the execution time is observed on two different computers with different configuration for 20 times. The first for each tool is recorded separately as the first run of each tools take longer time. The first run of the test codes usually indicates a fresh set of test codes. If there are no changes in the test codes, the later test runs take less time to run the codes. If there is any change that is implemented later the test run time increases as the tests run run from scratches. We have ran 20 more tests with each tool and find the average time needed for each tool to complete the test. To avoid any other impacts we ran all the test in single test file.

For hardware we chose one desktop configuration running Linux operating system and one laptop configuration running MacOS system. It would help us to identify if the operating system can cause different performance result. Also, during the development phase, software developers either uses desktop or laptop, so it was also our objective if different hardware configuration can bias the testing performance. Table 2 contains the detail of our two test machines.

Table 2: Hardware Configuration.

Item	Configuration 1	Configuration 2
CPU	4 Core 8 Thread	2 Core 4 Thread
Frequency	4Ghz	2.7Ghz
RAM	16 GB	16 GB
OS	Linux	Mac OS

First we ran the test on our desktop configuration. Tests were run for 21 times for each tool. From the records of Table 3 we can see on the first run Jest took the least time of 3.85 seconds and followed by Enzyme and Cypress with score of 5.62 seconds and 6.36 seconds. Then we ran the tests for another 20 times to find the average time for each tool.

In first configuration the Enzyme took least time to complete each 9 tests with an average of 3.73 seconds

Table 3: Test Scores on Configuration 1.

Test	Jest	Enzyme	Cypress
Initial	7.723	10.896	9.61
Average	5.25	3.73	8.12

followed by Jest with 5.25 seconds. Cypress achieved 3rd position with comparatively longer time of 8.12 seconds.

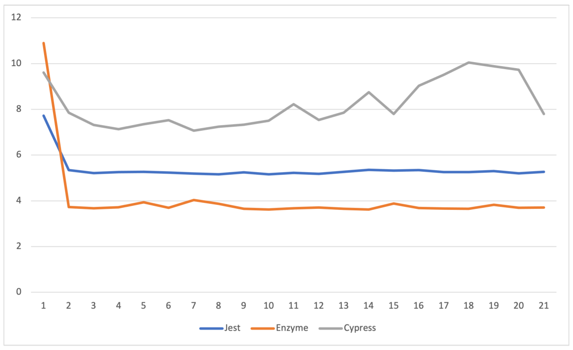

Figure 2: Test Run Time on Configuration 1.

Figure 2 shows that throughout the tests, the performance of Jest remains almost consistent including the initial test run. Enzyme behaves similarly, however the initial run time for Enzyme is almost 3 seconds longer than Jest. Cypress on the other hand is not as consistent as other two tools and provides different time in each tests.

Then we ran the same tests on our 2nd configuration and observed the performance of each tool. Since this configuration is comparatively weak, tools took comparatively longer time than 1st configuration. In such configuration, for the first run, Enzyme scored the lowest with 19.55 seconds, followed by Cypress and Jest with 18.81 and 16.016 seconds respectively. In consecutive tests, Enzyme scored first with average of 7.24 seconds, followed by Jest with 10.21 seconds and Cypress scored last with 14.70 seconds.

Table 4: Test Scores on Configuration 2.

Test	Jest	Enzyme	Cypress
Initial	16.016	19.55	18.81
Average of 20	10.21	7.24	14.70

In our configuration 2, Enzyme again scored the lowest score with 7.24 seconds, Jest finished the tests with an average of 10.21 seconds and Cypress scored last with 14.70 seconds.

Figure 3 indicates that both Jest and Enzyme performs similarly across the whole test runs. The output from 1st configuration and configuration 2 are identical, In 2nd configuration the time for each time is increased proportionally due to lower CPU configu-

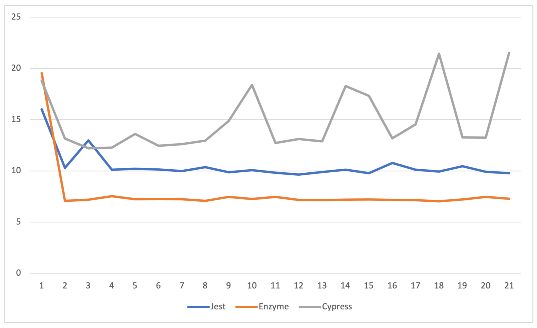

Figure 3: Test Run Time on Configuration 2.

ration. However comparing the Figure 2 & Figure 3 we can see the performance of each tool remains similar.

4.1 Observations

While implementing and analyzing each tool for testing, we observed and studied different scenarios for each tools. Functionalities like, tool scope, integration with project, developer friendliness etc were monitored.

- Enzyme is faster than other two tools in executing tests. Which is very helpful for performing testing with lots of tests, where testing takes less time allowing developers to deploy the application faster.
- Cypress is very easy to write and test. It also has native GUI to simulate the test activities. It is very easy for developers to begin with GUI testing. However, integrating Cypress is little bit complex, it also has OS library dependency in certain OS which increases the project setup complexity.
- Jest while is not as fast as Enzyme, in React applications, it comes built in, therefore there is no need for extra setup. Another big benefit of Jest is that it can be used for both back-end and front-end testing, making it very convenient for Full Stack developers.

From the analysis we found from our experiment, we generated a score for each tool, based on likert chart, where 1 is the lowest score and 5 being the highest. For Navtive GUI option we either gave 0 or 1 based on its availability.

From the Table 5 we can see Jest scores highest with 17 points, while Enzyme is very close with 16 points but Cypress is lagging behind with only 12 points.

Table 5: Findings based on analysis.

Research Topic	Jest	Enzyme	Cypress
Time	4	5	2
Integration	5	3	2
Ease of Use	3	3	5
Consistency	5	5	2
Native GUI	0	0	1
Total Score	17	16	12

5 CONCLUSIONS

As modern development is becoming advanced and more complex, more focus should be given to testing. For Front-End, as single page application development is getting popular day be day, Modern development practices such CI/CI is highly dependent on automatic processes, therefore an automated testing tool can contribute in reducing development and deployment time. We evaluated three different but popular testing tools and provided an empirical review.

This study was not empirically evaluated due to some unavoidable circumstances, such as covid situation was major. Besides, research questions (RQs) and analysis are solely based on literature reviews as well as work experience. However, analysis after doing survey or getting gap amongst the tools could have enriched the result more perfectly wherein some suggestions or guideline could be recommended for the testers to do automated testing more effectively and efficiently. This study compared the tools focusing on primarily execution time to perform the test cases. However, adding other factors to our study like resource usage could also improve our research with cost optimization. In future we plan to do a survey among different companies about their GUI testing tool usage for React based applications to find more industry relevant benefits and limitations and explore other parameters which might have contribution to testing performance.

REFERENCES

(2021). Front-end web development. Page Version ID: 1053051857.

2021. Front-end frameworks popularity (React, Vue, Angular and Svelte).

Aho, P., Buijs, G., Akin, A., Senturk, S., Ricos, F. P., de Gouw, S., and Vos, T. (2021). Applying Scriptless Test Automation on Web Applications from the Financial Sector. In *JISBD2021*. SISTEDES.

Aho, P. and Vos, T. (2018). Challenges in automated testing through graphical user interface. In *2018 IEEE Inter-*

national Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 118–121.

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

Alégroth, E., Karlsson, A., and Radway, A. (2018). Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 172–181.

Borges, N. P., Rau, J., and Zeller, A. (2020). Speeding up gui testing by on-device test generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1340–1343.

Broer Bahaweres, R., Oktaviani, E., Kesuma Wardhani, L., Hermadi, I., Suroso, A., Permana Solihin, I., and Arkeman, Y. (2020). Behavior-driven development (bdd) cucumber katalon for automation gui testing case cura and swag labs. In *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pages 87–92.

Chen, Y., Pandey, M., Song, J. Y., Lasecki, W. S., and Oney, S. (2020). Improving crowd-supported gui testing with structural guidance. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–13, New York, NY, USA. Association for Computing Machinery.

Dobslaw, F., Feldt, R., Michaelsson, D., Haar, P., de Oliveira Neto, F., and Torkar, R. (2019). Estimating return on investment for gui test automation frameworks. pages 271–282.

Garousi, V., Afzal, W., Çağlar, A., Işık, u. B., Baydan, B., Çaylak, S., Boyraz, A. Z., Yolaçan, B., and Herkiloğlu, K. (2017). Comparing automated visual gui testing tools: An industrial case study. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, A-TEST 2017, page 21–28, New York, NY, USA. Association for Computing Machinery.

Garousi, V. and Yildirim, E. (2018). Introducing automated gui testing and observing its benefits: An industrial case study in the context of law-practice management software. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 138–145.

Hasselknippe, K. F. and Li, J. (2017). A novel tool for automatic gui layout testing. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 695–700.

Iyama, M., Kirinuki, H., Tanno, H., and Kurabayashi, T. (2018). Automatically generating test scripts for gui testing. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 146–150.

Klammer, C. and Ramler, R. (2017a). A journey from manual testing to automated test generation in an industry project. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 591–592.

Klammer, C. and Ramler, R. (2017b). A journey from manual testing to automated test generation in an industry project. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 591–592.

Macchi, F., Rosin, P., Mervi, J. M., and Turchet, L. (2021). Image-based approaches for automating gui testing of interactive web-based applications. In *2021 28th Conference of Open Innovations Association (FRUCT)*, pages 278–285.

Moreira, R., Paiva, A., Nabuco, M., and Memon, A. (2017). Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*, 27.

Quental, N. C., de Albuquerque Siebra, C., Quintino, J. P., Florentin, F., da Silva, F. Q. B., and de Medeiros Santos, A. L. (2019). Automating gui response time measurements in mobile and web applications. In *Proceedings of the 14th International Workshop on Automation of Software Test*, AST '19, page 35–41. IEEE Press.

Silistre, A., Kilincceker, O., Belli, F., Challenger, M., and Kardas, G. (2020). Models in graphical user interface testing: Study design. In *2020 Turkish National Software Engineering Symposium (UYMS)*, pages 1–6.

Walkinshaw, N. (2020). Improving automated gui testing by learning to avoid infeasible tests. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 107–114.

White, T. D., Fraser, G., and Brown, G. J. (2019). Improving random gui testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 307–317, New York, NY, USA. Association for Computing Machinery.

Xiao-Fang Qi, Zi-Yuan Wang, J.-Q. M. P. W. (2017). Automated testing of web applications using combinatorial strategies. *Journal of Computer Science and Technology*, 32(1):199.

Xie, Q. and Memon, A. (2006). Studying the characteristics of a good gui test suite. pages 159 – 168.