



IMPLEMENTING UI REGRESSION TESTING PROCESS TO EXISTING WEB APPLICATION

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering, Master's thesis

2022

Miikka Lahtinen

Examiners: Professor Kari Smolander

M.Sc. (Tech) Timo Storhammar

ABSTRACT

Lappeenranta–Lahti University of Technology LUT
LUT School of Engineering Science
Software Engineering

Miikka Lahtinen

Implementing UI regression testing process to existing web application

Master's thesis

2022

45 pages, 2 figures, 3 tables

Examiners: Professor Kari Smolander

M.Sc. (Tech) Timo Storhammar

Keywords: Regression testing, regression, user interfaces, Selenium, JavaScript, ASP.NET

Software has become increasingly complex and has increased the probability of regressions when changes are made. With the rise of web-based applications quality control and more rigorous testing has become important part of the development cycle. This thesis investigates suitable UI regression testing frameworks and designs a testing pipeline for a company that has an established CI/CD pipeline and unit tests. This interest to automate regression testing stemmed from increased workloads and constraints caused by lack of available resources. Different frameworks were evaluated against requirements gathered from meetings and against existing software ecosystem. Pipeline as an artifact of the design science process introduced within this thesis reduces the amount of needed manual regression testing during development and deployment.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Miikka Lahtinen

Käyttöliittymän regressiotestaus prosessin implementointi olemassa olevalle web applikaatiolle

Tietotekniikan diplomityö

45 sivua, 2 kuvaa, 3 taulukkoa

Tarkastajat: Professori Kari Smolander

Diplomi-insinööri Timo Storhammar

Avainsanat: Regressiotestaus, regressio, käyttöliittymät, Selenium, JavaScript, ASP.NET

Ohjelmistot ovat muuttuneet entistä monimutkaisemmiksi mikä on lisännyt muutosten aiheuttamia regressioita. Verkkosovellusten myötä ohjelmistojen laadun ja jatkuvan testauksen tarve kehityssyklin aikana on kasvanut. Tämä paperi tutki yhteensopivia käyttöliittymien regressiotestaukseen frameworkkeja ja suunnitteli testaus prosessin yritykselle jolla on olemassa oleva CI/CD prosessi ja yksikkötestejä. Kiinnostus automatisoituun regressiotestaukseen syntyi lisääntyneestä työtaakasta sekä resurssien vähyydestä. Työ vertaili eri frameworkkeja sekä tapaamisissa saatuihin vaatimuksiin että olemassa olevaan kehitysympäristöön. Työn tuloksena oli prosessi joka vähentää manuaalisen regressiotestauksen määrää kehityksen ja uuden version julkaisemisen jälkeen.

SYMBOLS AND ABBREVIATIONS

| | |
|------|-----------------------------------|
| API | Application Programming Interface |
| BDD | Behaviour Driven Development |
| CD | Continuous Deployment |
| CI | Continuous Integration |
| GUI | Graphical User Interface |
| HTML | HyperText Markup Language |
| JSON | JavaScript Object Notation |
| SQL | Structured Query Language |
| TDD | Test-Driven Development |
| TUI | Text User Interface |
| UI | User Interface |

Table of contents

Abstract

Symbols and abbreviations

| | |
|---|----|
| 1. Introduction | 3 |
| 1.1 Background | 4 |
| 1.2 Goals | 4 |
| 2. Research methodology | 6 |
| 2.1 Design science..... | 6 |
| 2.2 Data gathering | 7 |
| 3. Software testing | 8 |
| 3.1 Regression | 10 |
| 3.2 Regression testing | 11 |
| 3.3 Regression testing challenges | 12 |
| 3.4 Regression testing techniques | 13 |
| 3.5 User interfaces..... | 14 |
| 3.6 Testability of user interfaces | 15 |
| 4. Solution design | 17 |
| 4.1 Existing development environment..... | 17 |
| 4.1.1 Desire to quality | 19 |
| 4.2 Initial requirements | 19 |
| 4.2.1 Constraints | 19 |
| 4.2.2 Framework | 20 |
| 4.2.3 Integrability | 20 |
| 4.2.4 Browser support | 21 |
| 4.2.5 Extra tools | 21 |
| 4.2.6 Popular and stable | 21 |
| 4.3 Frameworks..... | 22 |
| 4.3.1 Cypress..... | 22 |
| 4.3.2 Cucumber | 23 |
| 4.3.3 Mocha..... | 23 |
| 4.3.4 Gauge | 23 |

| | | |
|-------|---|----|
| 4.3.5 | Robot Framework..... | 24 |
| 4.3.6 | Selenium..... | 24 |
| 4.3.7 | CodedUI..... | 25 |
| 4.4 | Summary and selection | 26 |
| 4.5 | Pipeline integration | 27 |
| 4.6 | Process modifications..... | 28 |
| 4.6.1 | Prototyping..... | 28 |
| 4.6.2 | Prototypes to classes | 29 |
| 4.6.3 | Test structure..... | 29 |
| 4.6.4 | Tests in new features..... | 30 |
| 4.6.5 | Continuous integration and deployment | 30 |
| 4.7 | Evaluation and summary..... | 32 |
| 5. | Discussions | 33 |
| 5.1 | Reflections | 33 |
| 5.2 | Expansion of testing..... | 33 |
| 5.3 | Further development | 34 |
| 5.4 | Importance of regression testing | 35 |
| 6. | Conclusions | 37 |
| 7. | References | 38 |

1. Introduction

This thesis is a response to a request by a small company, which has decided to stay anonymous and will be referred just as the company from now on, is a market leader in its own field, which is electricity trading and portfolio management, offering software-as-a-service (SaaS) services for its customers. As the amount of software issues have been in decline over the years the company has expressed the desire to improve the testing quality as regressions still occur albeit less often. As the codebase is highly modulated and includes customer specific functionality and the fact that not every customer has the same features enabled, and the number of different modules makes testing them a chore every time version testing begins.

Regression testing as a term is somewhat masked behind normal cyclical testing during development. While unit testing, integration testing and system testing are more common terms regression testing has become equally or more important term in software development today. In a sense continuously running unit tests is one form of regression testing (Ali et al, 2019). Regression testing is defined as consistently running tests to avoid introducing regressions when a change has been made (Kapfhammer, 2011), which can include but are not limited to degraded performance, undefined behaviour, or completely broken features.

UI testing is often neglected in the software development due to it being difficult to implement. As more and more applications become web based where their user interfaces are accessible through browsers instead of the OS's own windows, the UI testing has become easier and more uniform across different browsers. Allowing single framework to handle the automation within web pages and bringing testing from purely JavaScript to other programming languages.

This thesis will attempt to investigate and analyze most fitting UI testing framework for the company specified in next chapter, lay the groundwork for the UI regression testing, define, and integrate new processes to handle UI testing. Following these the thesis will be similarly structured.

1.1 Background

Testing in the company is still mostly manual, while unit tests do exist and have caught regressions here and there, they haven't been expanded in a long while. However, expansion of unit testing has been planned to occur eventually but will require major refactors in most parts of the codebase to ensure that code is more test friendly which means possibility of running parts of program without needing database attached to the it. As part of that expansion, it was also decided to include regression testing, starting from user interfaces. Since the testing is mostly manual, the amount of time wasted in just checking if pages open, checking if certain critical features still work and other mundane navigation is quite large and diverts resources from testing the actual new features, these mundane tasks are also highly repetitive and necessary before and after deployment to production environment.

This thesis concentrates on looking on industry practices on regression testing while focusing on the UI to narrow the scope. It also aims to implement UI testing process into the existing pipeline to keep new software deployments more consistent and higher quality than previously possible. Thesis will also investigate and implement necessary changes to existing design processes by taking into the account of required software design changes when creating specifications for new features and pages. In general, the change should be minimally invasive and should be easily integrated.

As the artifact from this thesis is a proper test pipeline it's being created with the cooperation with testers and other necessary employees to understand what parts of the program and what test cases take priority in the initial implementation. Since scope of this thesis and the artifacts created from it won't cover entire program detecting the requirements and what initial implementation should include is crucial.

1.2 Goals

This thesis includes three goals. First goal that must be met is comparing and analysing different UI testing frameworks and select the most fitting testing framework to company's current tooling's which will be then implemented. Second goal is to lay the groundwork and implement a testing process that integrates into the already existing processes. This new

process will include all the necessary parts for future work on UI testing by including a library of methods attempting to reduce time needed for developing tests. Last goal will be mainly performing the retrospective to the created process and how it compares to the initial requirements.

2. Research methodology

This chapter introduces research methodology utilised in this thesis and explains how it was chosen and why. It also defines the data gathering methods and explains all the necessary information that is required for this thesis.

2.1 Design science

Design science is in its core a problem-solving research method with the aim of creating something concrete in application domain. For research to be design science it requires few key components, identification, solution, development, evaluation, addition of value and explaining the implications in practice. (March, 2008). Design science is especially common in information system research as it attempts to combine technology and social parts of information systems, code, and people. (Knutas, et.al, 2019)

Design science consists of an iterative process with seven steps with a possibility to fallback to previous steps if the goal of the step was not met. In order these steps are problem identification, definition of objectives, design and development, demonstration, evaluation and possibly communication if the solution is academically significant. (Peppers et. al., 2007)

Based on the definition of design science, it was chosen to be used within this thesis, meeting the need for creating artifacts in the form of documented process and concrete testing pipeline and framework. The artifacts themselves will be created for the company and therefore a design already exists, and these new artifacts must work with pre-existing frameworks, pipelines, and other architecture.

This thesis follows almost all the steps in design science. Problem that was identified was the number of mundane tasks performed prior and after new deployment. Objective was defined during the meetings and ended up being an automated UI testing pipeline that would work alongside existing pipelines and tools used by the company. Design changed as new information on which frameworks were suitable was uncovered, finally settling into Selenium and then designing the implementation around it. Implementation followed soon

after with a demonstration of proof-of-concept with the automation of most common tasks that users perform while operating the application, such as opening all pages. Evaluation hasn't finished at the time of writing as assessment of quality change is still ongoing as technical hiccups have hindered the progress.

2.2 Data gathering

Data required for the design science to be possible is gathered from meetings and initial specifications gathered from them. Apart from those the possible design was open-ended and free to implement. The initiative for implementing user interface testing came from an internally set goal to improve product quality to reduce amount of bug reports from customers and make version deployments smoother while reducing the amount of manual work prior and after deployment.

During the initial meeting, high level requirements for the testing framework were specified. This meeting also included a tester and a development manager to identify necessary features derived from requirements and wishes gathered from two different sides. During this meeting past attempts on expanding the testing to include user interface testing was also expressed. At this stage there were no preferred tooling or frameworks.

Subsequent meetings held roughly two weeks apart refined and introduced more requests regarding the process and compiled test cases that would be implemented in the first iteration of the implantation but also more comprehensive cases for future development. These helped to conceptualize what needed to be implemented initially, what kind of technology is required, what functionality should the underlying test project contain.

Eventually these meetings were phased out and integrated into already existing biweekly meetings to track the progress. Test cases were integrated into a separate issue ticket, which includes the actual implementable tickets and bugs related to that implementation ticket so that progress can be tracked.

3. Software testing

Software in general is a complex web of interconnected modules where single change in one of the modules can have subtle effects on other modules, possibly unintentionally breaking the existing functionality. Therefore, software must be tested to ensure that functionalities do match the specifications agreed prior to development.

Software testing is traditionally categorised into three distinct categories with each broadening the scale and each attempting to expose different kind of faults (Wahl, 1999). Unit testing is the most granular form of testing which consists of tests for single parts of a module to ensure individual functions do perform as expected. These tests tend to be numerous as they are often easy to implement and don't need to take other parts and functions into consideration.

Integration testing is a step up from unit testing and concentrates on modules as a whole and their interactions with each other as they would in the software itself. The aim of these tests is to discover inconsistencies and issues with these integrations. Ideally you would test each combination but often this tends to not be feasible due to number of modules. In addition, the number of stubs, blanket implementations of needed modules, is often high requiring sufficient development resources (Jeron et. al, 1999). This however is often a one-time investment.

System testing is the final testing form which is performed by utilising the program through the various systems it consists of. This testing is often done through the user interface by interacting with various created elements. If program is command line based, then every possible option and configuration must be tested and validated.

Ideally all tests would be run when a new change would be introduced into the codebase during development. This however is not always feasible. Software tends to grow naturally and sometimes sprawl increasing complexity of the software. As complexity of the software increases so does the number of runnable tests. This will easily lead to testing requiring a significant portion of developmental resources.

To combat these limiting constraints one possible methodology could be test modulation. In this methodology only a portion of tests would be executed focusing the efforts to the change itself and possible integrations it has with other modules. (Parson, 2014)

Generally, a software consists of a backend and frontend. Backend handles the lower-level functionality required for the frontend to operate and provides frontend with the information it should display. Modern software, especially web-based applications, has become increasingly frontend reliant and operations previously performed in backend are often migrated into frontend and verifying its correctness is increasingly more important. (Memon & Soffa, 2003)

When long term data storage is required, programs usually employ either a file-based storage systems or cloud-based storage often in form of a cache to speed up various operations, for example browsers cache image files to reduce the amount of network calls being made on subsequent loadings of a web page. Programs meant for enterprise settings and larger multi-user systems require more sophisticated data storage which often comes in form of a database. Poorly designed database schemas can hinder programs performance immensely and wouldn't necessarily be caught through traditional software testing. Different queries create different dependencies between tables within the database as well as different triggers if created (Haraty, et al. 2001) Performance tests would then be implemented to update the database schema to a more performant one, based on the current access patterns.

Regression testing which is the repeated act of software testing utilises these categories to its advantage by meticulously running tests whenever changes are performed to the codebase to find regressions or tests that worked on previous development cycle but were regressed after changes were made (Harrold, 2008). Finding regressions however are not always easy and can be easily missed, often being obscure requiring very specific conditions. Larger the codebase and module count within it the likelihood of a regression occurring grows. Large regressions are usually caught during the development or testing phases, but it's always possible for a small regression somewhere in the codebase to cause errors or invalid behaviour in completely irrelevant location making locating more time-consuming.

Main benefits from having comprehensive test suite that covers most if not all the lines of code are the reduced costs of maintenance later in the software development cycle. It also aids in reducing failure rates when performing cyclical releases where new version is

released every set period. Agile methods have short cycles where software is being tested often (Harrold, 2008) Software testing also aids the programmer to see if their contributions to the codebase cause any of the existing tests to fail, reducing the amount of time required to perform code review and speeding up the development cycle if care was taken during design and implementation phases.

However even when all tests are running all the time, even regression testing cannot find all underlying bugs and issues within the program. Achieving full code coverage is possible but quite often not feasible to execute and maintain due to the size of the program and other restrictions. Similarly, every new line of code will reduce coverage and requires a new test to be created to cover. Creating the initial test suite is a massive effort, often as big as the project itself if not larger (Labuschagne et al., 2017). Even worse, the amount of time required to perform all tests all the time increases as program grows, assuming that new tests are created as changes are made. This makes software testing not feasible to be performed all the time, at least not fully, since each modification must be tested, and testing would consume more resources compared to actual development. Software of today can often be configured by the user increasing the number of possible variants for each test (Robinson, 2011).

Cost and time are two main disadvantages that software testing has but can be circumvented somewhat. Reducing time can be achieved by modulating the tests that only test the module and its integrations where the change occurred or just reducing the number of times when tests are ran by prioritising the location within the process (Bhasin, 2014). These workarounds will also reduce costs by reducing the amount of computing resources required to perform software testing. Cost and time have a dependence between each other and reducing one often will reduce other. This optimization problem has been widely researched and multiple different mathematical solutions and technical methodologies have been proposed previously (Traon, 2000, Gupta 1996, Bhasin, 2014).

3.1 Regression

Regression in the context of software development means that functionality has changed because of a change brought to either the functionality itself or one of the dependencies

which can include modules, functions, and other locations on the codebase where changed functionality was called from. Regression is never a good change, resulting in including but not limited to degraded performance, software bugs, completely broken features and other possible scenarios that are too numerous to be listed since the number of variations are specific to the regression and underlying code logic. Regressions can become expensive as they trigger a fix and retest cycle which depending on the size of the codebase can consume majority of the allocated testing budget (Harrold, 2008)

There are various reasons why regressions happen, like inadequate code reviews, confusing and complex software architecture and code, but the most common reason is usually the lack of consistent testing or complete lack of tests. This becomes more apparent when multiple modules or a single module that is utilised in multiple contexts see changes to their code logic as part of either a new feature development or bug fixing. Similarly, programs that allow user to make configuration changes or programs that have customer specific configurations provide multitude of possible points of regression (Robinson & Qu, 2011).

3.2 Regression testing

Modern applications tend to have rapid development cycles as companies utilise agile methods instead of the traditional methods like waterfall method. Therefore, codebase sees changes often and likelihood for regressions increases. To keep customers satisfied with the application it's crucial to keep these regressions to absolute minimum. Unit tests are often created to prevent these regressions from occurring but depending on the architecture of the software it can be difficult or even outright impossible. New software projects can be planned to utilise TDD (test-driven development) methodology which prioritises test creation prior to actual development to begun. With the growth of agile methods and increasing reliance on CI and CD processes regression testing has become more prevalent and nature of regression testing has changed (Parsons, 2014).

Regression testing is the act of creating and running tests in such way that previously implemented functionality still performs as expected even after new changes are being applied to them. Prevention of re-emerging previously fixed software bugs is the main goal of the regression testing. Tests are usually small test cases with specific arguments which

initially replicated the software bug and later became guards against them. Initial difficulties can stem from initial sequence of inputs that triggered the bug. Creating a simplified and reduced test case that has no outside dependencies is the key for testability and future regression proofing. The test suite can contain different types of tests, progressive that test specification changes and corrective that test rest. (Parsons, 2014)

Regression testing is performed to avoid breaking or degrading changes and is often done by running unit tests regularly in attempt to catch failures before they would be introduced into the program. Ideally the test suite, which contains all tests of a project, covers every single part and every single combination of possible inputs there could be but is most of the time unfeasible to achieve. Therefore, prioritisation of test cases and input combinations is required to lower the resource requirements of testing.

Different parts of the application require different regression testing techniques. User interfaces are constantly being tested by the users and testers using different inputs. However, if the application has few users or testing occurs randomly the data might not be reliable (Sutapa et. al., 2020).

Databases are a core component of modern software for any meaningful long term data storage with majority of the systems using SQL (Haraty, 2001). In general SQL is the main language of database implementations but each implementation has their own quirks resulting in almost but not quite standardisation despite having ISO-9075 standard with newest standard defined in 2016.

3.3 Regression testing challenges

Regression testing is not always a straightforward process. Applications tend to have many integrations to third party libraries and code which most of the time are closed and there is no access to the underlying logic when creating tests. These challenges can be split into two categories, method, and organisation. Method related challenges would be handling failure types, performance measurements, handling fault distribution, toy examples and tooling whilst organisation related would be existence of structured tests, information availability, skills of employees and finally management support (Brahneborg et. al, 2017).

Modern software is increasingly parallel, leading to problems like data races which can be difficult to detect and create tests against. Attempting to develop tests that would cover these scenarios could end up flaky. Flaky tests are tests that occasionally fail or succeed when they are meant to result in the opposite value. Testing interfaces can also lead to multiple flaky tests since modern web is shifting towards more asynchronous functionality compared to the synchronous nature it used to be. Unpredictability of how long an element would take to show, especially if the functionality of said element is tied to a third-party integration, is a difficult challenge to overcome without introducing synchronicity. Flaky tests can emerge from tests that are dependent from each other and must be executed in serial as opposed to parallel (Lam, 2020)

With the rise of web-based technologies and modern UI frameworks the amount of configurability has increased significantly. Allowing users to configure their own systems creates more possible code paths that would need to be tested. It also creates more scenarios resulting in more different tests often with almost similar inputs differing by just one value (Robinson, 2011).

Testing can also eventually stagnate, or the initial technological hurdle is too great to even begin to test. This is often considered to be the cause of lack of tools, incorrect techniques or even lack of relevant data sets (Kapfhammer, 2011).

3.4 Regression testing techniques

Regression testing can be done in multitude of ways, and different techniques have been researched on their effectiveness to alleviate some of the regression testing challenges but also to alleviate the burden of lack of time related resources during software testing. In general, four different main categories can be found, these are retest-all, test selection, test prioritization and lastly test suite reduction (Rothermel, 2004).

Retest-all is the most expensive option time wise, running every single test all the time testing every part of the program that the test suite covers. Running all the tests when the introduced change is small is often waste of resources (Wahl, 1999).

Test selection focuses on running tests that have higher relevance to the change that was introduced. This method is particularly useful when testing is time constrained (Harrold, 2008). Selection is a dynamic process if needed, new test cases could be added if already selected set doesn't fulfill the criteria (Wahl, 1999).

Test prioritization is like test selection, but instead of attempting to select some of the tests to be run, test prioritization attempts to run critical tests first so that testing doesn't necessarily need to run all the way. Research has proven that test prioritization can result in faster fault detection and faster code coverage (Elbaum, 2003).

Finally, the test suite reduction, it is the last option where tests are removed from the test suite. These tests are often ones that fail often or are cosmetic tests or other tests that don't test actual functionality. As more tests are created and added to the test suite it often improves effectiveness at the cost of possibly overlapping previous tests making them redundant and obsolete. (Harrold, 2008)

Techniques mentioned in previous paragraphs are however not the only ones as there is no single general technique that excels over the others. Thousands of papers have been written about regression testing and its techniques (Ali et al., 2019). Different studies not only have different evaluation techniques but also unique artifacts (Engström et al., 2010).

3.5 User interfaces

User interfaces or UI for short are the user facing part of every program. Modern frontend frameworks are comprehensive and when used correctly can create visually impressive and flexible presentations without compromising functionality and compatibility between different platforms. This however comes at a cost of amount of code required to be written which can lead to a situation where majority of the products codebase is taken by UI implementation (Memon & Soffa, 2003). User interfaces are often deeply connected to different systems and servers and tend to include multiple layers to handle them which has become more prevalent in modern web applications. (Elbaum et. al, 2003)

From its roots of being simple text-based command line interface to modern rich, interactive interfaces. The development has seen few shifts, first from command line to graphical

frameworks and today a shift to web-based frameworks, further standardising the core logic allowing more automation.

Whilst command line interface or CLI is still very much alive and sometimes preferred, more on Linux systems than on Windows but also crucial on server management where having a proper GUI would be waste of resources. Another group that has been around before proper GUI's were introduced is TUI, a text user interface, which in a sense is a graphical interface but consists entirely of characters and can be drawn on terminal.

3.6 Testability of user interfaces

GUIs, or graphical user interfaces for short, have historically been difficult to implement testing for, as programs don't usually have any meaningful API to run automated tests on them, nor consistent or standardised methods. Automated solutions do exist and can be utilised for this purpose but require large amounts of setup for them to work consistently and even then, would be highly specific to one program. With the rise of web applications and having all the user interface in the form of a website, JavaScript and its derivative frameworks like React or Angular being the most popular frameworks alongside standardised HTML have created an environment where specific blocks of UI can be tested easily and more importantly consistently. However, due to the tightly coupled nature of a website to the server host, applications, especially web applications, tend to have their architectures modelled in multiple layers, requiring significant amount of work to setup a testing environment. (Elbaum et. al., 2003)

Another issue is with how often interfaces change, if a company decides to redo the user interface the work spent for creating previous UI tests is wasted. Any change to the interface, no matter how small, can change the usability of a test (Memon & Soffa, 2003). This alone can deter developers and companies from implementing test suites for UI's. While the initial workload is significant the technical debt that arises from old and complex interfaces and architectural issues is usually a bigger priority to solve as those affect performance of the application directly compared to the UI. Similarly interface rarely sees a complete overhaul as the amount of required developmental resources can freeze other processes until it is done

and tends to gravitate towards more modular growth unless proper processes and accurate and consistent designs are in place. This however creates a maintenance problem as one part of program is newer than others.

As mentioned in previous chapter user interfaces are often deeply connected to other services. This tends to create situations where the information presented to the user can change making consistent test cases harder to create. Ensuring a consistent and non-changing environment would require a custom environment which would be minimally updated. Applications reliant on databases would also require either updating the mock-ups or their own databases as new versions are released, and interfaces are changed.

Testing user interfaces is usually done by input recording software that can then play same inputs again. These tools don't usually require extensive knowledge on the underlying codebase and can be performed by testers. This method however can end up producing brittle tests that require rerecording every so often due to changes to UI, creating burden to the testers and taking resources out of testing proper (Vila, 2017).

Modern applications are increasingly web based where the interface itself is a website created using JavaScript frameworks. This standardisation of usage of HTML elements has allowed testing to be done consistently either through JavaScript or through the web browser using WebDrivers. Webdriver allows external applications to control the browser and access the webpage data through it, allowing automation of testing using different frameworks that normally wouldn't be able to perform UI testing. Webpages can also be presented differently; it can be provided by the server itself in the form of server-side renders or rendered directly by browser running the embedded JavaScript and HTML (Kuk, 2008).

4. Solution design

This chapter outlines the design of the proposed solution for implementing regression testing routine, concentrating initially on user interfaces. It also outlines necessary process changes to existing development process and outlines necessary requirements for the implementation.

4.1 Existing development environment

At the time of writing the company employs multitude of different tools to enable their current development processes. Issue management and tracking is done by using Jira. Each project is subject to version control using Git and are stored and managed in their own repositories on Bitbucket.

Current continuous integration and deployment process goes through multiple steps and multiple service providers. These being the Atlassian's Bitbucket, Microsoft's Azure and Octopus. As the pull request is made in the Bitbucket to one of the main branches, production, and next version, it triggers an automated build on Azure DevOps which handles the building of the software and running the unit tests within the project. The unit tests are not run when the pull request is made from a feature or bugfix branch but instead when the pull request comes from the other next version branch.

Each day the newest build of the new version of the project is deployed to a test domain for testing purposes. Testing at the time writing this paper is still manual if excluding the unit tests run explained in previous paragraph. This increases the testing times tremendously and even simple tasks such as opening each page to make sure that they still function as they should takes up a large chunk of a tester's worktime. Therefore, it is imperative to optimize at least some of the more mundane and repeatable tasks by utilizing automated tests.

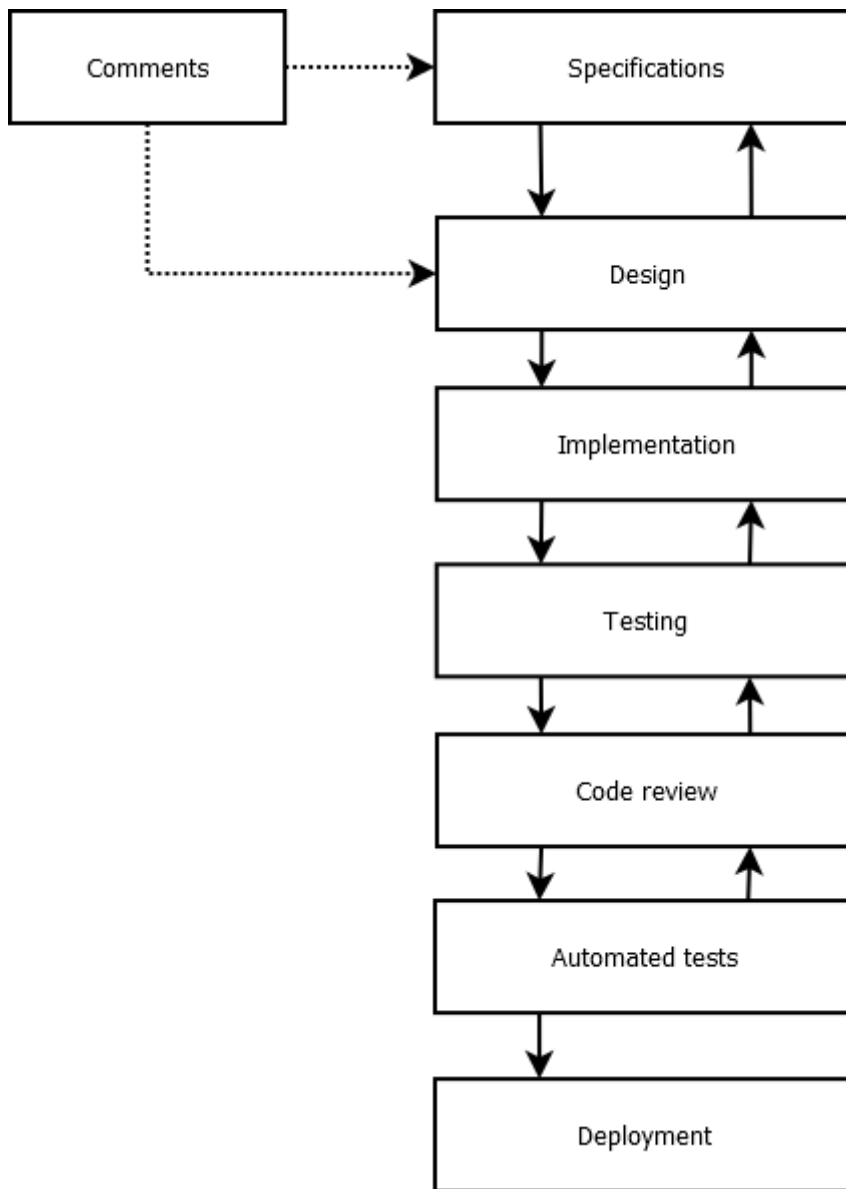


Figure 1: Current development cycle

Figure 1 shows the flow of the existing development pipeline that the company currently employs. It is a rather straightforward but is flexible enough to allow mistakes in design and implementation to be corrected. Figure also does not include all possible transitions between states nor extra states prior to implementation, these extra states do exist to allow commenting on the written specifications or design. It also shows the post development phases with code reviews and automated testing prior of being accepted and merged into the destination branch.

4.1.1 Desire to quality

For a while the company has had a desire to reduce the amount of bug reports, from internal and external sources. There has been previous work on implementing and automating unit tests and that work continues to evolve, but majority of testing is still repetitive and not resource efficient. Prior to deployment testing is done manually on the parts that are not covered by unit tests and were touched on during feature development, and often ends up being resource heavy, time and development wise and this problem is further worsened by the small size of the company. After new version has been deployed a handful of steps are performed to ensure that actual deployment went smoothly, and server and program are running as expected and no critical regressions happened.

Previously created unit tests do function and have on occasion prevented regressions, but these tests only cover a small portion of the program. Since the architecture of the program doesn't lend well to testability the effort to expand these tests has been big enough to justify the delay of the implementation.

4.2 Initial requirements

This chapter will concretely list few of the requirements acquired through meetings mentioned in chapter 2.1. It is not an exhaustive list as the solution is meant to be more of an expedition than a full implementation. This was done due to small size of the company and limited developer resources. Therefore, gradually introducing the development team into regression testing and UI testing itself.

4.2.1 Constraints

This implementation will not be an extensive implementation with the aim of testing entire project nor a generic implementation that can be deployed for multiple different projects that share same base programming language. Instead, this implementation will concentrate on

few hot paths that not only are often tested but are also most important ones from customers point of view and can be easily automated without significant effort.

4.2.2 Framework

While there weren't many restrictions on what frameworks and what other tools could be chosen there were however some limitations introduced during the meetings mentioned in previous chapter. Since company does employ testers who doesn't necessarily have experience in software development and knowledge of programming, a tool that would allow rapid prototyping and easier and faster way of recording test cases to be then eventually properly integrated into actual code-based test cases was preferred. Since company already utilised xUnit as part of their unit testing efforts there was an incentive to utilise same framework as developers were more familiar with it. In a nutshell the framework chosen which will be utilised within the solution has to accommodate current developers' familiarity with previously used frameworks.

4.2.3 Integrability

Expanding on the previous requirement since majority of the company's products utilise C# as their main programming language and every developer is familiar with it. JavaScript was another prominent language so frameworks from these two programming languages was preferred not only from familiarity point of view but also from easier maintainability point of view. Therefore, this solution must consider what current developers are familiar with it and how it would fit to current projects. Similarly, the solution must be able to be easily integrated into the existing CI/CD pipeline for testing to be automated instead of relying on manually running the test suite.

4.2.4 Browser support

Today's browser market is primarily dominated by three different browsers. Chrome, Firefox, and Safari. Microsoft also has Edge, which is Chromium based, Chromium being the browser engine used in Chrome and many desktop JavaScript programs of today almost holding a monopoly. Despite all three being similar by their support of modern web technologies there are minor differences and thus it's a requirement that the selected framework to support multiple browsers. This would allow detection of changes that would have no issues on one environment but would cause issues or not work at all on another.

4.2.5 Extra tools

Since the company is on the smaller side and utilisation of manual labour for most tasks takes up a significant portion of resources timewise an inclusion of something extra that would not necessarily remove manual part but would speed up the work. Smaller the company more tasks are shared between multiple people to allow the flexibility if one member of the group is on leave or falls ill. Initially there were two options for the possible solution to investigate. First one was a higher-level scripting language that would not require much programming knowledge but would still allow flexibility through more natural programming language. Second one was a tool or tools, browser extensions or separate, that would allow testers to build their own tests and repeat them without requiring programming knowledge for example through recording their own inputs as they navigate the program. First option was eventually dropped and second one was made a non-mandatory requirement but was stated to be a nice to have. Solution is not required to have one, but this requirement is considered a benefit.

4.2.6 Popular and stable

Another major requirement was that the selected framework should not be either too new or too old and would have to be widely utilised within the software development projects. The

selected framework must also be stable and include a good documentation to aid developers who are not previously familiar with the framework.

4.3 Frameworks

This chapter explores the different UI testing frameworks and compares them to the requirements to the defined requirements while also considering any non-requirements that each framework might offer. Criteria are the backbone of any selection and will be defined before even attempting to select frameworks.

Multiple frameworks were taken into consideration, and they are separated in two categories. Purely JavaScript based and non-JavaScript based frameworks. There are multitude of different automation testing frameworks on JavaScript, and it makes hard to keep the number of selected frameworks down. The software products of the company are all web applications and therefore can be tested utilising JavaScript based UI testing frameworks.

When researching possible non-JavaScript based frameworks the selection tends to be quite limited when taken C# support into consideration. While there are plenty of unit testing frameworks that could possibly be utilised for testing the UI, it would require much more implementation and would be wildly out of scope of this thesis.

Eventually few testing frameworks were identified as possible candidates from each category, from JavaScript side there are Cypress, Jest and Mocha. And from the non-JavaScript side Robot Framework, Selenium, Microsoft's CodedUI, Gauge and Cucumber. Frameworks that support multitude of different programming languages and therefore have their own language specific packages are considered as non-JavaScript.

4.3.1 Cypress

Cypress is a popular end-to-end testing framework for web applications. It advertises itself as a Selenium alternative and aims to support modern JavaScript frameworks such as React, Vue and Angular where support for Selenium would be lacklustre or difficult to implement. The tests are written in pure JavaScript.

Cypress while being an open-source project is not entirely free. It includes a free pricing plan but is quite limited on the number of tests it can run before hitting the limits of the free plan. Of course, the paid plans and the enterprise plans do come with an email support instead of relying on the community for support.

4.3.2 Cucumber

Cucumber is a popular BDD-based testing framework which aims to have a more natural language as the language for tests is relatively close to English. While Cucumber is not aimed to be a UI testing framework it can work in conjunction with Selenium to allow its tests to be ran on websites, and thus doesn't come with its own test recorder. Close to natural language it is also easier than many other frameworks that expect programming knowledge. Cucumber also has good integration with Azure pipelines.

4.3.3 Mocha

Mocha is a popular JavaScript based open-source testing framework which can run either on node.js or in browser which lends well to testing websites one by one. Mocha supports modern JavaScript standards and features a rich feature set. This framework is extremely popular within JavaScript projects. Mocha's architecture attempts to replicate a more natural flow of code which when read would be close to English allowing user to follow code easier.

4.3.4 Gauge

Gauge is test automation framework that has its tests written in Markdown, making it stand out from most other frameworks which rely either on created language of their own or any number of more standard programming languages. UI testing is available through another package called Taiko which Gauge developers suggest using.

4.3.5 Robot Framework

Robot Framework is python based generic automation framework, which utilises their own language by default, it doesn't support user interface testing, but it can be added through external libraries. Like Cucumber, Robot Framework too was not aimed towards UI testing but as like Cucumber it can be implemented through Selenium. Therefore, it doesn't have a test recorder of its own. Robot Framework being a Finnish creation is quite popular amongst Finnish companies such as Nokia and Finnair.

4.3.6 Selenium

Selenium is a cross platform automation test framework that has been ported to multiple languages and utilises web drivers to control the browsers. It's one of the most popular frameworks to drive browsers from non-JavaScript environments. At the time of writing, Selenium supports multiple different languages such as Python, C#, and Java. Selenium has also developed a browser extension for Firefox and Chrome that allows quicker development of test cases and allows categorization and management of said tests.

Selenium's main component which allows it to communicate with various browsers is called a web driver. Web drivers are the interface between Selenium and browser, translating the calls from Selenium into commands supported by the browser. These web drivers are configurable to the point where each web driver could have their own profiles and configurations, allowing greater flexibility to test different use cases with different configuration options.

According to W3C, which is the regulatory body of internet standards, the web driver is a standard that is assumed to be equal between all modern internet browsers. The standard specifies how information from the browser is presented and how it can be accessed through the web driver. It defines how each element in a webpage is presented and methods for running JavaScript and basic functionality within the browser. Navigational controls and other necessary common web development features such as cookies, prompts and screenshots are also specified.

Selenium offers a web browser extension called Selenium IDE as an option for easier test development to those who either do not want to write program code or to those who want prototype tests and run them locally without setting up a test environment. This extension is supported in multiple browsers and offers similar feature set as the actual framework, with ability to run JavaScript and customize timeouts for waiting for HTML elements. Extension also has plenty of predefined actions and locator methods built in.

Extension also allows exporting of the testcases as code classes for different testing frameworks such as xUnit and NUnit. These are fully functional, but relatively simple. The file that these test cases are stored on is JSON based file that can be ran using Selenium Grid. Custom runner could also be implemented to run tests defined within the file but would require significant effort. This file can be transferred between browsers to be ran anywhere at any time. Allowing sharing of the file and saving it to the version control as a backup should something happen is a bonus.

4.3.7 CodedUI

CodedUI tests is a Microsoft specific feature within Visual Studio. It enables the UI testing within the projects. It does support plenty of different application types from web applications to windows applications. However, as of writing this this feature will be deprecated and will no longer be included in newer versions of Visual Studio, leaving the Visual Studio 2019 as the last version to include this feature. Microsoft does suggest to either utilizing Selenium for web applications and Appium with a windows application driver for desktop usage on UWP applications.

Like Selenium it also offers a recorder for test cases but is limited on the playback if used in any other browser than Microsoft's own IE and Edge when it comes to browsers. Overall, it is a Microsoft specific user interface testing tool.

4.4 Summary and selection

After doing initial evaluation of presented frameworks, it is quite interesting how many non-JavaScript frameworks rely on Selenium itself to drive their browser tests. JavaScript based frameworks have the advantage since despite many of them having server integrations through Node.JS they are still written in the same standard HTML as all webpages and modern web technology-based applications.

Table 1: JavaScript based frameworks

| Feature | Cypress | Mocha |
|-----------------|----------------|--------------|
| Language | JS | JS |
| Azure support | Yes | Yes |
| Browser support | All | All |
| Recorder | No | No |
| Popular | Yes | Yes |

Table 2: non-JavaScript based frameworks

| Feature | Robot Framework | Selenium | CodedUI |
|-----------------|------------------------|-----------------|------------------------|
| Language | Python/Own | Multiple | C# |
| Azure support | Yes | Yes | Yes |
| Browser support | All (Selenium) | All | All, with reservations |
| Recorder | No | Yes | Yes |
| Popular | Yes | Yes | ? |

Evaluation was done on each framework, weighing their positives and negatives regarding the specified requirements in chapter 4.2. This allowed for a realistic comparison between the company's needs and features of the frameworks. Despite being previously explored and later abandoned, Selenium stood out as the best framework that suited the needs of the company at present time.

Table 3: Frameworks with uncategorized language

| Feature | Gauge | Cucumber |
|-----------------|-------------|------------------|
| Language | Markdown | Own/English-like |
| Azure support | Yes | Yes |
| Browser support | All (Taiko) | All (Selenium) |
| Recorder | No | No |
| Popular | ? | Yes |

Company had attempted to use Selenium in the past but wasn't adopted back then, from same reasons as this time, lack of available resources which were better spent in other areas. Selenium provided a good and easy integration with company's existing pipelines and allowed utilisation of existing testing framework through xUnit. Selenium also offered a recording tool which turned out being not exactly easy to use but wasn't overtly difficult either for those who don't have programming knowledge and export functionality to xUnit classes was a bonus.

4.5 Pipeline integration

Company already employs a CI/CD pipeline through Microsoft Azure as their main way to automate the building and testing the software through automated unit testing. Selenium will be added as another part to this pipeline. UI testing pipeline will somewhat be a separate from the main pipeline as it was meant to be a daily or manually triggered task. Simplicity and ease of use were goals for this pipeline.

Azure DevOps is a highly configurable pipeline as variables can be defined manually or by other pipelines. Selenium will be implemented to take advantage of it through XML substitution and this is done to avoid having login credentials saved on the Git repository itself. Each C# project has their own configuration file that is XML based making it relatively easy to make tests to be ran on different URL's using different credentials should the need arise.

4.6 Process modifications

There are three distinct steps outlined within this process. While this process assumes that the process is followed in all cases, it can differ when the test creation is initiated by a designer who has created the specification for the feature or change or by the developer who has created it during development of said change. In these cases, the first two steps can be skipped as it is mainly targeted to UI elements that do already exist and tests are being created by a tester who has seen such test to be necessary to automate.

Parts of this process are to be integrated into the workflow itself, when possible, test cases should be considered not only for unit tests but to also UI tests where applicable. UI elements must have clear identification methods utilizing their id attributes which should be understandable yet compact to reduce the line length when inspecting the DOM through different tools. ASP.NET, which the company uses, creates these attributes but are often very obscure and can change from version to another, making creating a one specific test to a single element a hassle. However, it's doable utilizing CSS attribute selectors with their ability to partially match to an attribute or even XPath selectors using their absolute paths in the DOM.

4.6.1 Prototyping

Testers and developers who would utilize the Selenium IDE extension can record the tests which are then transferred to the person who is responsible for the further development and integration into codebase instead of test cases being on an extension to not only prevent flakiness on tests but to also reduce the amount of time needed to rerecord tests. It's highly favourable to annotate these tests and steps to ensure that others and test creator themselves can recognize the steps afterwards (Ricca, 2021)

These prototypes, or recordings are the first step and can be easily shared through the project files that Selenium IDE uses. These JSON based files contain all necessary information regarding the test cases, groupings and test scripts or actions.

4.6.2 Prototypes to classes

Selenium IDE has a feature where the test cases can be exported as a class file which could be integrated straight to the project. These tests, while working, are however not ready to be included on the project as is. Since the company's products have different modulations for different customers not all customers have same elements and pages. Similarly, these tests can often end up being flaky, where test cases would occasionally fail or would fail all the time if the order of elements changed, or elements were replaced. Therefore, it's imperative that these skeleton classes are forwarded to the developers either on the same issue ticket that also includes the change or on a separate issue ticket.

Some things are also needed to take into consideration, while testers might operate on environments with enough computational resources, the tests are aimed at customer specific features maybe in a specific environment and a specific database behind said environment. There are no logging or error handling most of the time, unless the tester is proficient with programming, which cannot be taken for granted. Often error messages are cryptic to an untrained person and will cause extra work for developers.

4.6.3 Test structure

Like unit testing where each test corresponds to a single module and functionality within, user interface regression testing can be performed to single elements within a page. Whilst the capability for cross-page testing is possible e.g., testing page traversals, it's lesser objective to single page approach.

Following the popular testing frameworks, the tests will be modulated on a page level, in attempt to reduce the amount of testing needed to perform and to speed up the testing process through modulation. Some grouping will also be done to reduce the number of duplicated tests and code, aiding in future maintenance and development of the test libraries.

A library of more common functionalities will be provided in attempt to reduce the amount of time required to perform an action and to reduce the amount of code duplication within

the tests across the codebase. These functions won't provide any specialized functionality but rather act as a platform to build the test cases from.

While Selenium offers two different methods for selecting specific web elements, CSS selectors and XPath, maintaining XPath selectors which are fixed paths to the element from the root of the document is a larger effort than with CSS selectors. (Leotta et al., 2013)

4.6.4 Tests in new features

Integrating the step of designing new UI tests to the actual design phase is easier to perform than creating new tests to the already existing elements within the program and its interface. Same issue appears also with unit testing where it's easier to implement new tests to new functionality when the written code is new and feature specifications and design are fresh in memory. New and refactored code will benefit testing by making sure that code is easily testable without significant overhead and initial effort.

Ideally whenever new elements are introduced or old and outdated are expected to be updated through feature development or other refactoring work, specifications should also take testing into account and consider what nature of tests could be needed prior to actual design and implementation. Creation of new elements tend to often be just rough drafts up until implementation, making it hard to envision what should be tested and how.

4.6.5 Continuous integration and deployment

As mentioned in chapter 4.5 this implementation also includes changes to the existing pipeline. The UI testing pipeline introduced in this thesis will be automated to run daily when newest version is installed but will also allow triggering it manually should there be a need to test on new environment or the daily trigger failed. Since deployment also relies on Octopus Deploy there is no sure way to know when the deployment is finished, and deployed environment would be up. Therefore at least initially the UI testing pipeline will run on timer, but on long term a proper triggering system should be developed. Eventually a proper triggering system should be made to ensure that deployed environments are up to allow UI

tests to be run on that environment without either waiting for environment to wake up when first visiting or failing the tests due to possible timeouts, creating flaky tests.

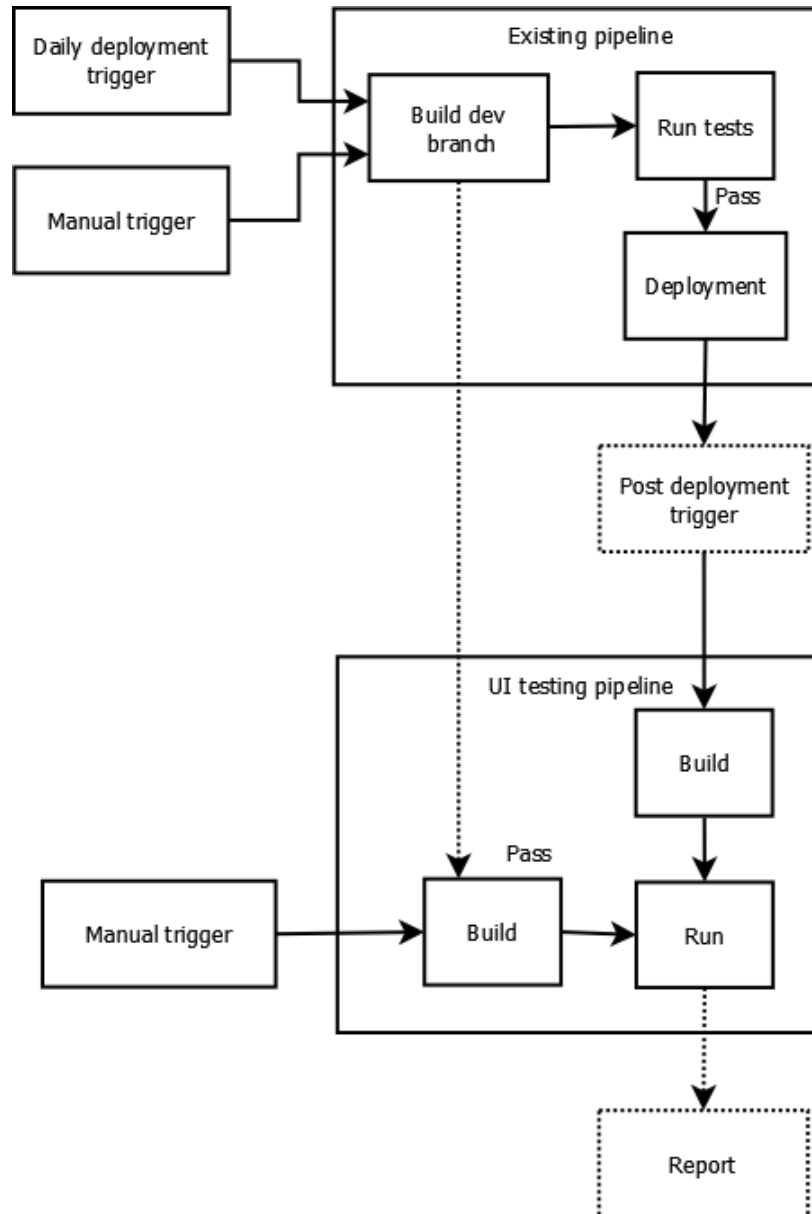


Figure 2: Existing and proposed UI testing pipeline

The developed UI testing pipeline described in figure 2 would be run after the main pipeline but can be run independently from it should a need arise. These needs could be broken configurations, broken tests or even testing new environments that have been manually created outside of the existing pipeline. Ideally it should be possible to trigger a testing pipeline through configuration so that the main project would not build and instead only testing pipeline would.

4.7 Evaluation and summary

Developing unit tests is still relatively underutilized within the company and is not prioritized during development which leads to plenty of modifications to existing practices and processes. Change naturally cannot be quick but for testing to take more priority a change in the work practice is needed. It is highly likely that development of these tests will be done by single or at maximum few persons in the short term.

As mentioned before testing was mostly manual for some simpler and core functionality prior and post deployment taking away resources from other areas. Pipeline introduced in previous chapters won't solve all of it but will alleviate large amounts of more mundane tasks that would've previously been manual.

Selenium which was selected as the framework to be implemented fits quite well to the company at its current state, being attempted in the past and now bringing it back for second attempt. Other frameworks would've required significant initial effort to be implemented as majority would require their own server environments. Selenium doesn't have an extensive library behind it but can be easily expanded by third party libraries, in a long run a custom library is needed that suits the company's products. As shown in table 2 it covers the initial requirements well but suffers from lack of functionality outside of its core functionality which concentrates more on WebDriver.

Overall, the proposed pipeline and addition of UI testing to counter regression is a positive effect on quality and stability between version releases. It relieves the burden that is put on testers since they can know ahead of time that a test deployment has already ran through the previously manual tasks and can concentrate on the new changes fully. It will take time for state of testing to improve considering the size of the company and its available resources and past practices.

5. Discussions

This chapter discusses of the current impact of the thesis and future development of regression testing. In addition, it will also discuss what can be done to improve the initial implementation.

5.1 Reflections

In the end this thesis attempted to answer the needs of a single company and may not be reflective on regression testing practices used in other companies. This thesis ended up being more of a scratch of a surface rather than a deep dive. When researching literature regarding regression testing it became clear how much research has already been done and how much there is still to research. Combined with ever changing programming techniques, languages, and tools there won't be a lack of areas needing more research.

Implementing the proposed pipeline was relatively straightforward process with one caveat. Every server has IP restrictions from everywhere else except their own office network. As the pipeline agents that perform the pipelines are Microsoft hosted their IP's change weekly making it difficult to open just one IP. This however can be worked around by utilising a separate VM that runs the UI tests. This brings additional maintenance work to keep browsers updated should new security patches arrive.

5.2 Expansion of testing

As the looming possibility of expansion beyond the company's home country is getting closer the requirement of having a robust testing suite has become almost a necessity. Testing process produced in this thesis and the already existing unit tests are a good enough of a platform to build upon. It is not however an extensive one and coverage of the program remains poor.

The existing process recently re-enabled the unit testing which had been dormant for a while. Mainly the reasoning behind this was that nobody wanted to maintain those tests. This on itself was a regression within the process where the entire pipeline was reworked to allow testing to be automated and reduce the amount of manual testing. The testing is still mostly manual due to requirements of having a database, assortment of customer specific features, files being required for importing data and many of the features not supporting the way tests are written

5.3 Further development

While this paper only concentrated on the initial implementation of the process and integration to the existing systems and processes there's more work to be done to allow further and deeper integration. In current state there's still need for conversion from recorded tests to actual test written into a test case in code.

Similarly, more work is required to the existing processes and pipelines to refine and further develop these to better suit more rigorous testing and testing multiple deployments simultaneously. While there is no timeline for this development work right now it should be taken into a serious consideration and implemented in small enough batches to be feasible even for a small team and to prevent any serious delays whilst providing a benefit of faster testing and more time for development.

At the time of writing this paper, there has been a resurgence of interest in expanding the unit test suite to include more, equally important parts of the program to ensure reliability in different cases. As the shift from customer-based code towards more unified handling progresses the code will be refactored to be more test friendly and thus expanding tests will require less resources.

This thesis did not touch the database or the functionality that utilizes said database aspect of the software. This was done to reduce the scope and concentrate on a single area of regression testing that was easier to implement. Introducing a regression testing to this area would take too long and take much of the company's resources as current functionality and code structure does not lend itself well to even unit testing. So as a suggestion is that the

expansion of unit testing to be done first before even thinking of expanding UI tests to concentrate the development resources to areas which need it more.

As mentioned, a refactor should happen slowly, new features should be designed with testing in mind while not being directly test-driven development. As older parts are touched during the development of newer features it is suggested to refactor them to work better with unit testing. There is a caveat however, attempting to refactor might end up being more work than the new feature would initially require.

Another expansion point would be improving the generalization for detecting changes when inputs to elements are given. This could be implemented through snapshot testing, where two snapshots which are small pieces of DOM string are taken before and after the input and then compared to each other, outputting what nodes changed and how. This method would allow quicker and simpler way to detect DOM changes when there is no need-to-know which element or what attribute changed but would allow the flexibility for filtering should it be needed.

5.4 Importance of regression testing

Regression testing is a highly researched area of software testing due to its inherent nature of becoming expensive, requiring different techniques to optimize and reduce the time constraints. (Garousi, 2016, Ali et al., 2019) This lends well to implementation of various mathematical models to describe possible solutions.

With increasingly complex and integrated software the prevalence of incorrect behavior increases dramatically and in today's world a single mistake in code can result to loss of life or loss of expensive scientific instruments. Software today is a service that customers buy which in turn means that it must be maintained and updated regularly, services can be introduced or phased out, parts of the program can become a bottleneck as number of users increase and, in some cases, especially with open source, users themselves report issues often perceiving their issues as critical for usability.

Agile methods brought challenges to software development that were not present previously, much faster development cycles meant customers and users would be more tightly coupled

to the development resulting in pressure to keep services running as smoothly as possible to prevent losing customers over performance or stability issues. Regression testing will become increasingly important as number of users increase or there is a change to any of the features.

6. Conclusions

Regression testing is the act of consistently running specified test cases to find regressions within the software. This thesis resulted in the framework and initial foundation for expanding the already existing testing procedures. Requirements were given through meetings and understanding the teams' available resources and competencies. Testing frameworks were then analysed and compared to find the most fitting one to the already existing development framework and existing processes with an aim to minimise the impact of introducing different programming languages to pre-existing development process.

Introduction of Selenium as the main UI testing framework brought interesting technical challenges, integration with pre-existing testing framework xUnit prevented of adding another dependency to the product where it was implemented. The pipeline also supports configuration for the Selenium should URL domains of the deployment sites change in the future.

The current state of tests remains limited, but easily expandable in the future. Future improvements could also include of having always available testing environment just for running unit and UI tests which would take always latest version from the versioning system when changes to the Git repository happens. Similarly, the workload to implement more tests through the products is large and will take time and resources to implement rest of the tests.

7. References

- Ali, N. bin, Engström, E., Taromirad, M., Mousavi, M.R., Minhas, N.M., Helgesson, D., Kunze, S., Varshosaz, M., 2019. On the search for industry-relevant regression testing research. *Empir Software Eng* 24, 2020–2055. <https://doi.org/10.1007/s10664-018-9670-1>
- Brahneborg, D., Afzal, W., Čaušević, A., 2017. A Pragmatic Perspective on Regression Testing Challenges, in: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). Presented at the 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 618–619. <https://doi.org/10.1109/QRS-C.2017.117>
- Bhasin, H., 2014. Cost-priority cognizant regression testing. *SIGSOFT Softw. Eng. Notes* 39, 1–7. <https://doi.org/10.1145/2597716.2597722>
- Elbaum, S., Karre, S., Rothermel, G., 2003. Improving web application testing with user session data, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03. IEEE Computer Society, USA, pp. 49–59.
- Engström, E., Runeson, P., Skoglund, M., 2010. A systematic review on regression test selection techniques. *Information and Software Technology* 52, 14–30. <https://doi.org/10.1016/j.infsof.2009.07.001>
- Garousi, V., Mäntylä, M.V., 2016. A systematic literature review of literature reviews in software testing. *Information and Software Technology* 80, 195–216. <https://doi.org/10.1016/j.infsof.2016.09.002>
- Hao, J., Mendes, E., 2006. Usage-based statistical testing of web applications, in: Proceedings of the 6th International Conference on Web Engineering, ICWE '06. Association for Computing Machinery, New York, NY, USA, pp. 17–24. <https://doi.org/10.1145/1145581.1145585>
- Haraty, R.A., Mansour, N., Daou, B., 2001. Regression testing of database applications, in: Proceedings of the 2001 ACM Symposium on Applied Computing, SAC '01. Association for Computing Machinery, New York, NY, USA, pp. 285–289. <https://doi.org/10.1145/372202.372342>

Harrold, M.J., Orso, A., 2008. Retesting software during development and maintenance, in: 2008 Frontiers of Software Maintenance. Presented at the 2008 Frontiers of Software Maintenance, pp. 99–108. <https://doi.org/10.1109/FOSM.2008.4659253>

Jeron, T., Jezequel, J.-, Traon, Y.L., Morel, P., 1999. Efficient strategies for integration and regression testing of OO systems, in: Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443). Presented at the Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443), pp. 260–269. <https://doi.org/10.1109/ISSRE.1999.809331>

Kapfhammer, G.M., 2011. Empirically Evaluating Regression Testing Techniques: Challenges, Solutions, and a Potential Way Forward, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. Presented at the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 99–102. <https://doi.org/10.1109/ICSTW.2011.88>

Knutas, A., Pourzolfaghar, Z., Helfert, M. (2019). The Role and Impact of Descriptive Theories in Creating Knowledge in Design Science. In: Holzinger, A., Silva, H., Helfert, M. (eds) Computer-Human Interaction Research and Applications. CHIRA 2017. Communications in Computer and Information Science, vol. 654. Springer, Cham. pp. 90-108. DOI: https://doi.org/10.1007/978-3-030-32965-5_5

Kuk, S.H., Kim, H.S., 2008. Automatic Generation of Testing Environments for Web Applications, in: 2008 International Conference on Computer Science and Software Engineering. Presented at the 2008 International Conference on Computer Science and Software Engineering, pp. 694–697. <https://doi.org/10.1109/CSSE.2008.1026>

Labuschagne, A., Inozemtseva, L., Holmes, R., 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017. Association for Computing Machinery, New York, NY, USA, pp. 821–830. <https://doi.org/10.1145/3106237.3106288>

Lam, W., Shi, A., Oei, R., Zhang, S., Ernst, M.D., Xie, T., 2020. Dependent-test-aware regression testing techniques, in: Proceedings of the 29th ACM SIGSOFT International

Symposium on Software Testing and Analysis, ISSTA 2020. Association for Computing Machinery, New York, NY, USA, pp. 298–311. <https://doi.org/10.1145/3395363.3397364>

Leotta, M., Clerissi, D., Ricca, F., Spadaro, C., 2013. Comparing the maintainability of selenium WebDriver test suites employing different locators: a case study, in: Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation, JAMAICA 2013. Association for Computing Machinery, New York, NY, USA, pp. 53–58. <https://doi.org/10.1145/2489280.2489284>

March, Storey, 2008. Design Science in the Information Systems Discipline: An Introduction to the Special Issue on Design Science Research. MIS Quarterly 32, 725. <https://doi.org/10.2307/25148869>

Memon, A.M., Soffa, M.L., 2003. Regression testing of GUIs, in: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11. Association for Computing Machinery, New York, NY, USA, pp. 118–127. <https://doi.org/10.1145/940071.940088>

Parsons, D., Susnjak, T., Lange, M., 2014. Influences on regression testing strategies in agile software development environments. Software Qual J 22, 717–739. <https://doi.org/10.1007/s11219-013-9225-z>

Peppers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S., 2007. A Design Science Research Methodology for Information Systems Research. Journal of Management Information Systems 24, 45–77. <https://doi.org/10.2753/MIS0742-1222240302>

Ricca, F., Leotta, M., 2021. Towards automated generation of PO-based WebDriver test suites from Selenium IDE recordings, in: Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation. Association for Computing Machinery, New York, NY, USA, pp. 9–16.

Robinson, B., Qu, X., 2011. Customer-Oriented Regression Testing: An Initial Discussion, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. Presented at the 2011 IEEE Fourth International Conference on

Software Testing, Verification and Validation Workshops, pp. 107–110.

<https://doi.org/10.1109/ICSTW.2011.99>

Shariff, S.M., Li, H., Bezemer, C.-P., Hassan, A.E., Nguyen, T.H.D., Flora, P., 2019.

Improving the testing efficiency of selenium-based load tests, in: Proceedings of the 14th International Workshop on Automation of Software Test, AST '19. IEEE Press, Montreal, Quebec, Canada, pp. 14–20. <https://doi.org/10.1109/AST.2019.00008>

Sutapa, F.A.K.P.G., Kusumawardani, S.S., Adhistya, E.P., 2020. A Review of Automated Testing Approach for Software Regression Testing. IOP Conference Series. Materials Science and Engineering 846. <http://dx.doi.org/10.1088/1757-899X/846/1/012042>

Vila, E., Novakova, G., Todorova, D., 2017. Automation Testing Framework for Web Applications with Selenium WebDriver: Opportunities and Threats, in: Proceedings of the International Conference on Advances in Image Processing, ICAIP 2017. Association for Computing Machinery, New York, NY, USA, pp. 144–150.

<https://doi.org/10.1145/3133264.3133300>

Wahl, N.J., 1999. An overview of regression testing. SIGSOFT Softw. Eng. Notes 24, 69–73. <https://doi.org/10.1145/308769.308>