Overview of Behaviour-Driven Development tools for web applications

Viktor Österholm

MSc in Computer Science
Åbo Akademi University
Faculty of Science and Engineering
Department of Information Technologies
May 2021
Supervisor: Dragos Truscan

Abstract

Testing is an important part of software engineering. To avoid failure, software needs to be tested to ensure that it fulfils the requirements and that it is as defect-free as possible. Additionally, software needs to help the customer reach their goals, such as gaining a competitive advantage, launching a new product or feature, etc. The goal of Behaviour-Driven Development (BDD) is to help software development teams deliver highly valuable, high-quality software faster. BDD focuses on close customer interaction and writing tests based on scenarios before the code is written.

The goal of the thesis was to help software development teams choose the correct BDD tool for developing and testing web applications, as choosing the wrong testing tool can be time-consuming and costly. The thesis compared three testing tools that can be used for BDD: Robot Framework, Behave (the Python port of Cucumber), and Concordion. The tools were compared based on 10 questions within five categories that were found to be important to testers when choosing a testing tool. To help evaluate the tools, a test case was implemented on a web application. The assessment found that both Robot Framework and Cucumber are good choices for BDD for web applications, as they are popular, well-supported, and relatively easy to use. Concordion was less popular and harder to use due to a lack of learning resources. Nevertheless, Concordion was found to be suitable if flexible specifications are needed and the testers have good coding knowledge.

Keywords: Behaviour-Driven Development, Automated acceptance testing, Software testing, Web applications

Table of Contents

1. Introduction

Nowadays, software and web applications play a significant role in many businesses and people's lives. Web applications are used to, e.g. store and organise information, collaborate within teams, buy and sell items and stocks, watch movies, videos and tv-shows, and much more.

Creating successful applications is no easy task. Around half of all software projects fail to deliver in some way. [1, p. 4] A common reason for software projects being impaired and ultimately cancelled is that the requirements are incomplete. Many software projects are also challenged due to unrealistic expectations and unclear objectives. [2, p. 9] It is therefore clear that improving the requirements gathering and the communication between the stakeholders in the software project will increase the likelihood of the software project succeeding.

This is the primary goal of Behaviour Driven-Development (BDD), a relatively new software development methodology. BDD facilitates communication within teams to ensure that only the features that align with the stakeholders' business goals are developed. In BDD, the features are written as user stories that can be used to run acceptance tests using a testing tool that supports BDD. With BDD, the user story functions as documentation of the feature, as well as automated acceptance tests (AAT) that are used to validate that the feature fulfils the user's needs.

Many testing tools have been developed that can be used for BDD and automatic acceptance testing of web applications. Choosing which tool to adopt is important because switching tools can be costly, as it often requires training and big changes in the codebase. When adopting a testing tool, software development teams should consider many things, such as the price and features of the tool, the documentation and support of the tool, and how easy it is to use. This thesis will attempt to make this process easier by examining and comparing three testing tools that can be used for BDD: Cucumber, Robot Framework, and Concordion.

The thesis is structured as follows: In Chapter 2, different software development methodologies will be explained that provide background for the development of

agile, the software development methodology that is used with BDD. Chapter 3 provides a short overview of web applications. In Chapter 4, software testing will be explained. Chapter 5 explains BDD, the benefits and drawbacks of BDD, and the BDD development process. Chapter 6 provides an overview of the three testing tools. Chapter 7 lists ten questions within five categories that will be used to compare the tools. In the later sections of the chapter, a test case is also implemented for each tool that tests a popular web application. In Chapter 8, the testing tools are compared based on the ten questions. Chapter 9 summarises the results and provides suggestions for future research.

2. Software development methodologies

To understand testing and software development concepts that are discussed in this thesis, it is useful to understand how software development has evolved over the years. Of particular interest to this thesis are agile methods, as BDD is more commonly used in teams that use agile methods [1, p. 30], [3, p. 4]. It is therefore important to understand what agile is and how it came to be. Two of the most common software development models prior to agile were the waterfall model and the V-Model. Agile was developed because these software development models were seen as inadequate in many cases.

2.1 The waterfall model

One of the first software engineering methodologies was the waterfall model, which was introduced in the 1970s [4, p. 9]. The waterfall model dictates that software should be developed linearly, starting with a lengthy planning process, leading to implementation and ending with testing and verifying that the system works as intended. This is done in phases, starting with the requirements analysis phase and ending with the system maintenance phase.

During the requirements analysis phase, the requirements of the system are gathered from the client and documented in a requirements specification document. In the next phase, the system design is prepared. Hardware and software requirements such as programming languages to be used are specified in the system design. Using inputs from the system design, the system is then developed in small programs called units in the implementation phase. The units are tested for their functionality before being integrated in the next phase. In the testing and integration phase, the units are integrated into a system, after which the complete system is tested. In the deployment stage, the system is released to the market or deployed in the customer environment. During the maintenance phase, patches are released that fix errors in the system. Improved versions of the software might also be released to the customers. [5], [6] See Figure 1 below for an illustration of the phases in the waterfall model.

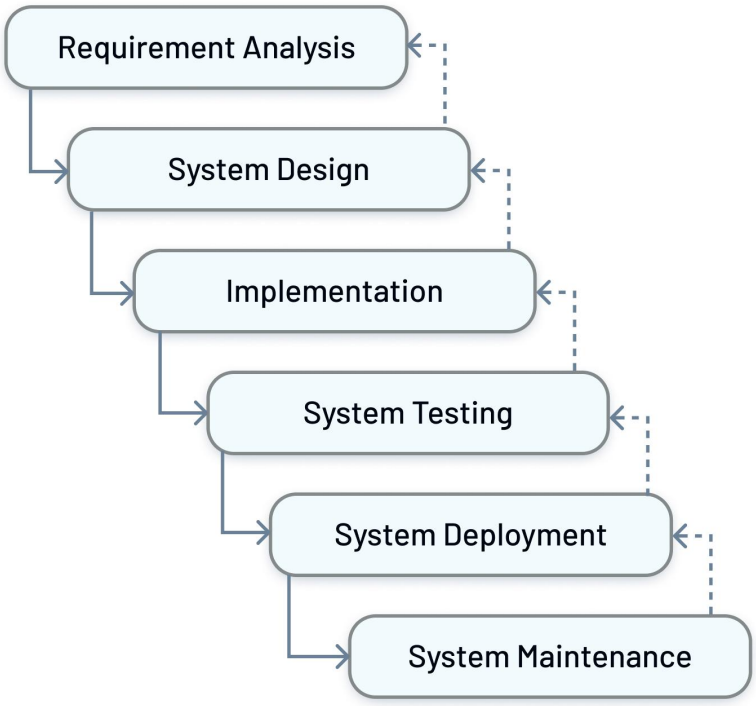

Figure 1: The waterfall software development model (adapted from [5], [6]).

Over time, it has become clear that this linear way of developing software rarely works well in practice. One of the biggest problems of the waterfall model is that it does not allow for changes to the plans in the middle of development. During a software project, there are often unforeseeable difficulties and changes to the requirements. The waterfall model is too lengthy and linear to allow significant changes within the project. Because testing is done last, it also leads to defects being discovered late in the process, which makes them expensive and difficult to fix. Increased competition has also led to the need to develop new features and fix bugs quickly, something that is not possible using the waterfall model.

2.2 The V-model

The V-model of software development is an extension of the waterfall model. In the V-model, each phase in the software design phase is mirrored by a testing phase. The phases of the left side of the "V" in the V-model are phases that are used for verification. The phases on the right side of the V-model are used for validating the

system using tests. The V-model is sequential, as it requires that the phases are completed in order. Figure 2 below shows the different phases in the V-model. [7]

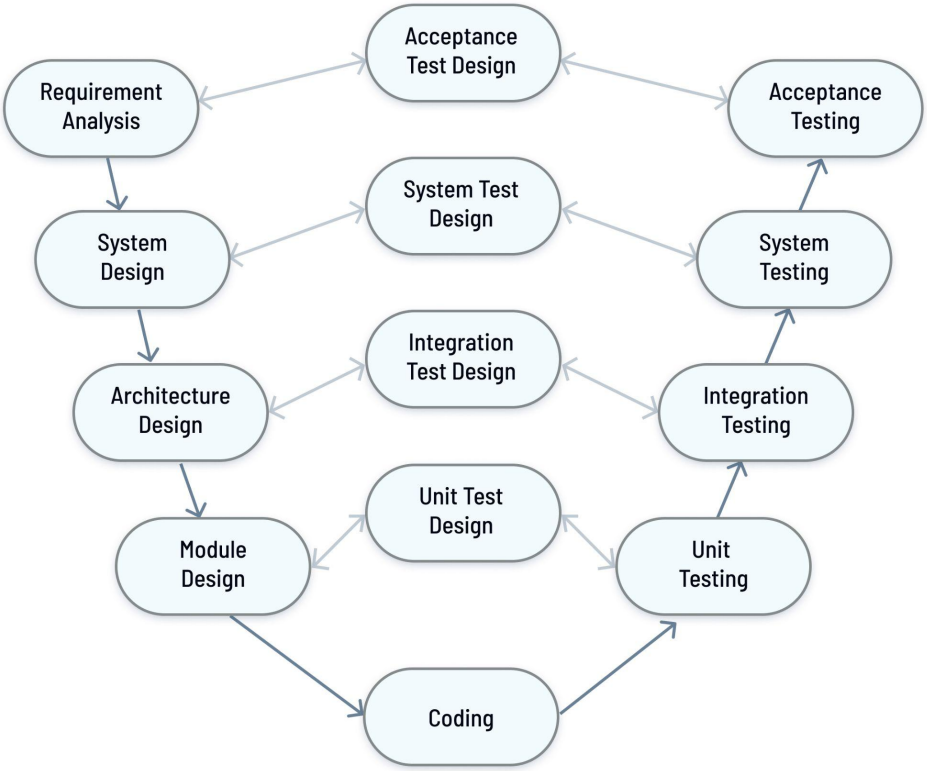

Figure 2: The phases of the V-model (adapted from [7]).

The verification side of the V-model also starts with the requirements analysis, and the system design phase. The system design phase is followed by the architectural design phase, where the system design is broken down into modules with different functionalities. The information flow between the modules and external systems is also defined during this phase. The final phase of the verification process is module design, where the internal design of the components in the system is specified. [7]

After each verification phase in the V-model, tests that correspond with the phase should be designed. After the verification phases are completed, the system should be implemented with code. This is illustrated by the middle part of the V-model. [8]

In the first phase of the validation side of the V-model, the small components of the system are tested. In the second phase, integration tests are carried out to test the communication between the parts in the system. During the next phase, the system as a whole is tested. The last phase is the acceptance testing phase, where the system is tested to ensure it satisfies the requirements specified by the customer in the requirements analysis phase. [7]

The V-model has many of the same drawbacks as the waterfall model; it is difficult to change the requirements or functionality later in the project. This rigidness makes the V-model less suitable for software development projects with changing requirements. Development of the software also starts late in the process, as much planning and design must be completed first. [7]

2.3 Agile

Because of the problems with the waterfall model discussed above, agile methodologies for software development were developed. Agile is an adaptive software development method, compared to traditional software development methods such as the V-model or the waterfall model that are predictive. Agile focuses on developing features rather than lengthy planning. To minimise the risk of major failures, the software is tested regularly during development. Close collaboration within the team and customer interaction is important in agile. [9] Agile is important to this thesis because many of the testing and software development methods mentioned in this thesis are only compatible with the agile model, or at least commonly used in agile teams.

There are many agile software development methodologies, all of which are based on the four values expressed in the Agile Manifesto and the 12 Principles behind it [10]. The four values expressed in the Agile Manifesto are [11]:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Scrum and Extreme Programming (XP), both agile frameworks, have influenced practices used in BDD [1, p. 262], [1, p. 95]. These frameworks will therefore be explained shortly in the following sections.

2.3.1 Scrum

Using Scrum, an agile process framework, the work is split into 2-4 week iterations called sprints. The goal of the sprint is to produce a potentially shippable product increment. Each sprint starts with planning, which occurs in two parts. In the first part the items to be worked on are chosen from the product backlog, creating the sprint backlog. In the second part, the team determines the tasks necessary to successfully deliver the items. Each day, the team has a short discussion to determine the activities for the next day. The entire team and the stakeholders participate in a sprint review and the end of the sprint. The results of the sprint are discussed and demonstrated. The stakeholders may also be given a chance to use and give feedback on the increment. Each sprint ends with a retrospective where the team reflects upon how things went and what adjustments could be made for the next iteration. [12]

2.3.2 Extreme programming

Agile Alliance defines Extreme Programming (XP) as “an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team”. Important values of XP include close communication within the team, simplicity, and constant feedback. XP is most suitable when the software development team is small, the project has dynamically changing software requirements, there are risks caused by using new technology, and it is possible to utilize automated unit and functional tests.

Some important practices in XP include involving the whole team in the development, using stories written from the user’s perspective that describe what the product should do, splitting development into small iterations, and writing tests before code is written. [13]

2.3.3 User stories

User stories are short sentences that describe who does what and why. A typical user story is of the format “As a (role) I want (functionality) so that (benefit)”. There are many reasons for expressing requirements as user stories. User stories do not focus on implementation details, instead, they convey the needs and wants. This is important as rarely care about how the software is built, but rather what value it delivers to them. Short stories are also good at keeping people’s attention compared to structured documents. If storytelling is not used, there is a higher tendency to mention technical implementation details, something that should be avoided at this stage. It is common to write down the user stories on cards during meetings with the customers, as this makes it possible to e.g. arrange the cards in order of importance. [14, p. 325-326]

3. Web Applications

As the tools in this thesis will be evaluated by testing a web application in Chapter 7, it is important to understand what a web application is, and the basic structure of web applications. According to [15, p. 5], a web application is “a client/server application that uses a Web browser as its client program, and performs an interactive service by connecting with servers over the Internet (or Intranet).”. Web applications are more dynamic than web sites, which mostly only serve static content [15, p. 5]. There are many similarities between native applications and web applications. For instance, web applications are focused on one task or topic and they provide an interface that can be used to interact with the application. [16, p. 19] Because web applications are accessed over the internet using a web browser, the same version of the web application can be accessed from many locations, by many users using many different devices and browsers, at the same time [17]. Some examples of web applications include Google Docs, Gmail, Evernote and Netflix.

Web applications consist of a client-side part and a server-side part. The client-side part is the part that the user sees and interacts with in the browser. The server-side part contains most of the business logic, and it can be seen as the “brains” of the application. [18] The server-side of web applications will not be explained further in this thesis, as it is not necessary to understand how the acceptance tests in Chapter 7 are implemented.

3.1 The client-side of web applications

The client-side of a web application contains the user interface (UI) of the web application and everything else the user can see in the browser, such as text and images. Some common user interface elements and controls include buttons, text and input fields, checkboxes and sliders. The client-side of a web application is implemented using HTML, CSS, and JavaScript, a programming language that is used to make web pages interactive.

HTML stands for HyperText Markup Language [19], a markup language that defines the structure of the web page. HTML-documents contain tags, also known as

elements, that are defined using opening and closing brackets. Below is an example of the markup of an HTML-element that defines a button:

```
<button>Click Here!</button>
```

HTML documents can be styled using Cascading Style Sheets (CSS) [20]. HTML elements can be styled by giving them classes or IDs that identify them on the web page. Using CSS, different styles can be applied for elements of a certain type, or for elements that have certain attributes, such as particular classes and IDs. Below is the markup of the same button, after class and ID attributes have been applied:

```
<button class="styled-button--red" id="unique-button">Click Here!</button>
```

JavaScript is a versatile programming language that can be used to e.g. add, modify, and delete HTML elements, run code in response to events performed on the page, such as mouse clicks, and much more. Below is an example of JavaScript code that finds an HTML element with the CSS ID “foo”, changes it to “bar”, and then displays the text content of the element inside an alert box [21]. The JavaScript code in the example has been embedded inside the body tag of the web page.

```
<body>
    <script>
        let element = document.querySelector("#foo");
        element.id = "bar";
        alert(element.textContent);
    </script>
</body>
```

4. Software testing

Software Testing is the process of checking that software fulfils the expected requirements and ensuring that software is defect-free. Using manual or automated testing tools, software or system components are executed to evaluate one or more properties of interest. Software testing is used to discover errors, which can originate from implementation or from missing, incorrect or incomplete requirements. [22] To do this, test cases are written, which give the system under test (SUT) some input. The output of the SUT is then compared to some expected output. If the output of the program matches the expected output, it passes the test case, otherwise, it fails the test case. Related test cases are commonly organized into test suites [23]. See Figure 3 below for an illustration of how a test case interacts with the SUT.

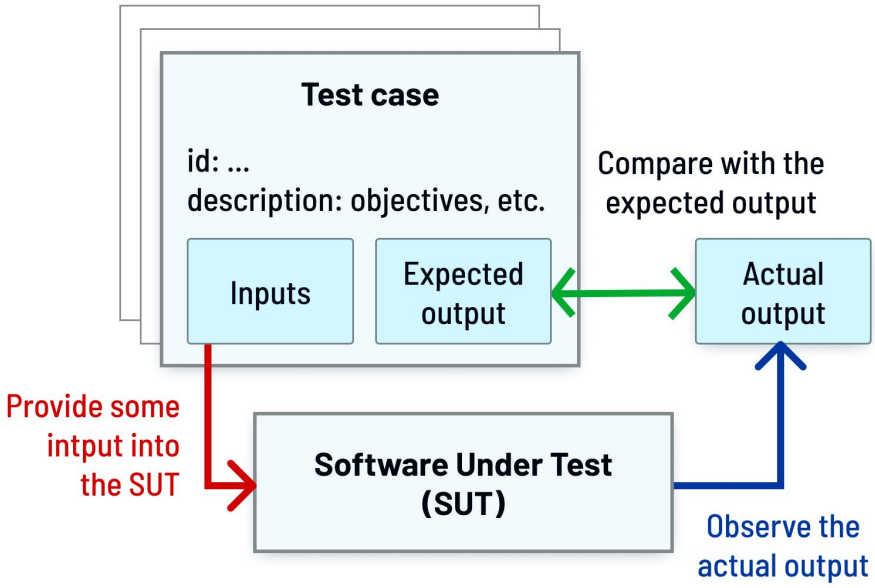

Figure 3: How a test works (adapted from [24]).

A test execution engine or testing framework can be used to automate the process of sending inputs to the system, comparing the output produced by the system with the expected output, and creating test reports [25]. Examples of testing frameworks include Jest for JavaScript, and JUnit for Java [26]. The test report might contain information such as how many test cases were run, and how many of the tests passed

and failed. Good test reports will make it clear what test cases failed and depending on the type of test, the size of the program, and if the source code is available, approximately on which line of code the error occurred. It is also preferable that the test reports are easy to read and understand for non-technical people, as the reports might be shared within the team.

4.1 Unit and integration testing

Unit tests are used to test that a part (e.g., component, module, function) of the system works as intended. Most of the code should be unit tested to catch bugs early. For these reasons, most of the tests should be unit tests. Since the goal of unit testing is to verify the functionality of individual modules [27], one needs to do integration testing to verify their interaction. Integration tests are tests that expose defects in the interaction between software modules that might be developed by different programmers.

4.2 System testing

System testing tests the software product as a whole to determine whether the system meets its specifications. System testing is typically done to discover specification and design problems. [28, p. 6] System tests typically test the application from a customer's perspective, treating the system as a black-box [29, p. 207]. There are more than 50 types of system tests, both functional and nonfunctional. For the latter, examples include load tests, which tests how the software performs under real-life loads, usability tests, which test how easy or difficult the system is to use, and regression tests, which test for the presence of software bugs during development [30].

4.3 Acceptance testing

Acceptance testing is one of the last types of testing to be done before the release of the software. The International Software Testing Qualifications Board defines acceptance testing as "Formal testing with respect to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria and to enable the user, customers or other authorised entities to determine

whether to accept the system." [31]. In short, acceptance testing is used to determine whether the software, or a part of it, meets the needs of the customer that were captured during the requirements analysis phase. Similarly to system testing, acceptance testing is also a type of black-box testing, meaning that the tests do not test the program code directly, but rather the system as a whole. Parts of the system such as the UI or the application programming interface, or reports produced by the system can also be tested with acceptance tests. [8, p. 51]. The benefits of acceptance testing is that acceptance tests can prove that the application works as intended in a real-world scenario and that it delivers the business value its users are expecting. [32, p. 189]

In order to fully simulate a user's experience when interacting with an application, the acceptance tests can be run against the UI of the application. This will test the same code paths that the users of the system will invoke in real interactions. However, this comes at a cost, as it is difficult to write maintainable acceptance tests that are coupled to the UI. This is because changes to the UI during development, and after release due to usability improvements, spell corrections, etc. are common. For instance, if acceptance tests contain a reference to a UI element and the UI element changes, it will break all the tests that reference the UI element. [32, p. 191]

4.4 Test automation

Test automation is important because manual testing can be extremely expensive in large software projects. If software needs to be tested manually before each release, the costs of releasing new software can be too high, preventing the company from releasing software frequently. [32, p. 188-189] Most types of tests can be automated using testing tools.

The test pyramid, which is illustrated below in Figure 5, provides a rule of thumb for implementing automated tests [33, p. 311-312]. It argues that there should be fewer UI tests than unit and integration tests. This is because writing UI tests that will remain valid and useful and that will not break easily takes longer, and they are harder to maintain. It is also not uncommon for teams to have acceptance tests that cannot be run every night because they take so long to run. [33]

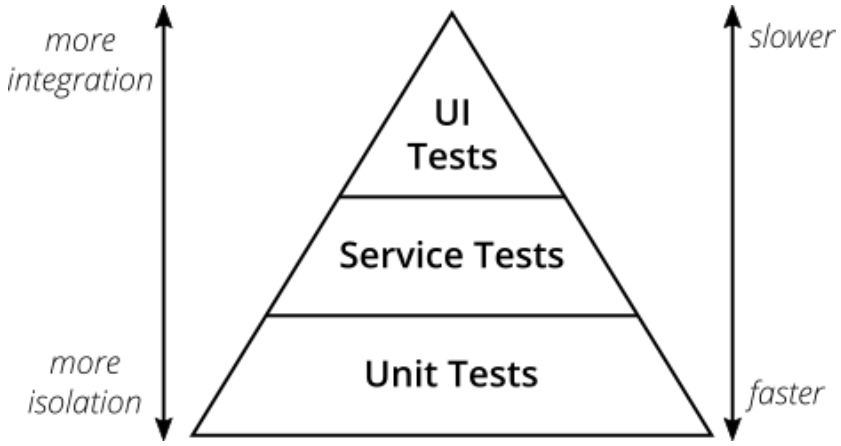

Figure 4: The test pyramid [34].

5. Behaviour-Driven Development

Behaviour-Driven Development (BDD) is a set of software engineering practices invented in the 2000s by Dan North. BDD is designed to help teams deliver higher quality software that delivers more value to the customer, faster. BDD is built on Agile practices, such as Test-Driven Development (TDD). According to [1, p. 12] "BDD provides a common language based on simple, structured sentences expressed in English (or in the native language of the stakeholders) that facilitate communication between project team members and business stakeholders".

BDD was invented as a way to teach TDD, a software engineering practice that uses unit tests to specify, verify, and design software code. While TDD has many advantages, it can often lead to developers focusing too much on the details, instead of the business value the code is supposed to provide. An example of this, is that the names of unit tests, whether they are written with TDD or not, are often not descriptive. For instance, if the name of a test only consists of the name of the method it is testing, plus the word "test", it becomes hard to understand, and fix the method if it breaks. North noticed that naming tests based on the functionality of the methods they are testing helps developers write more meaningful tests. By focusing on the expected behavior, instead of the method being tested, it becomes easier to focus one's efforts on the underlying business requirements. North also noticed that naming tests using full sentences, using the word "should", helps focus the attention

on the requirements. An example would be renaming the test *testTransfer()* to the more descriptive *should_transfer_funds_to_a_local_account()*. This makes the name of the test more similar to a specification than a unit test. Because tests written in this manner focus on the behaviour of the software, North started calling this approach *Behaviour*-Driven Development. [1, p. 12-14]

North and his colleague Chris Matts realized that this could be applied to the requirements analysis phase. They wanted to create a language that business analysts could use to define requirements, and that could also be used to run automated acceptance tests. Their solution was to express acceptance criteria for user stories using specifications that explain how the feature should work, also known as "scenarios". The scenarios make it clear what should be developed and tested. Below is an example of such a scenario: [1, p. 14]

```
Given a customer has a current account
When the customer transfers funds from this account to an overseas account
Then the funds should be deposited in the overseas account
And the transaction fee should be deducted from the current account
```

This notation is now commonly known as the Gherkin style. [1, p. 14] The emphasized words in the example above are Gherkin language keywords, the rest is written as natural language. Testing tools that support BDD are able to transform scenarios written in Gherkin into automated acceptance tests by associating the sentences with methods using regular expressions [3, p. 47].

5.1 Benefits and drawbacks of BDD

The main benefit of BDD is that it facilitates organized communication within the team. This means that the product owners, developers, and testers will have a better shared understanding of how the system works. With BDD there is a direct path from end-user requirements to usable, automatable tests. The requirements written by the customer in the Given-When-Then format can be directly used as the starting point for acceptance tests. This means that it is easier for non-developers to participate in the creation of the acceptance tests. Chapter 6 will explain in more

detail how requirements written in Gherkin language are implemented as executable acceptance tests.

Some drawbacks of BDD is that prior experience with TDD is needed to adopt it. BDD is also incompatible with the waterfall software development model, and it may not be effective if the requirements are not properly specified. Testers using BDD also need to have sufficient technical skills. [35]

5.2 The BDD development process

The BDD development process consists of three steps (see Figure 5). The first step is to create a user story for an upcoming feature and to discover how the system should behave using concrete examples. The user stories should be related to business outcomes. Only the behaviours that contribute directly to these business outcomes should be implemented. The second step is to document those examples in a way that could be used to run acceptance tests automatically. This is commonly accomplished using the Given-When-Then format. To improve communication, the domain experts, testers, and developers should have direct access to this documentation [36]. During the third step, the documented behaviour is implemented with code. The automated test is used to guide the development of the code. Over time, this process helps create living documentation, that is, documentation that accurately reflects the behaviour of the system [37, p. 16].

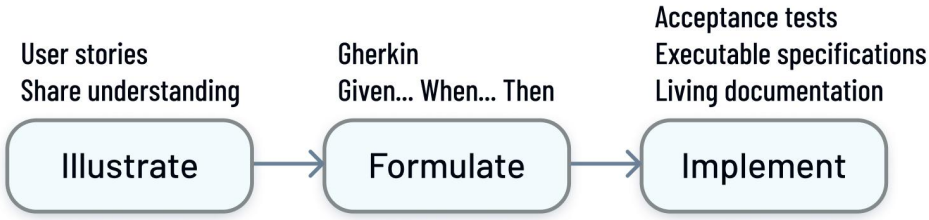

Figure 5: The BDD development process (adapted from [38]).

6. Overview of acceptance testing tools

Many tools that can be used for acceptance testing and automation testing have been developed. There are at least 59 tools that can be used for automated testing [39].

There are both commercial and open-source acceptance testing tools. Commercial tools often require less coding skills of the testers and they also have better support. Paid tools are often an all-in-one solution that can handle all types of testing. Open-source tools require some coding skills for writing and running the tests and they typically focus on one type of testing.

The following sections will provide an overview of the testing tools and walk through the implementation of a test on a simple to-do list application. The same test will be implemented with all the tools to show the exact implementation differences. The to-do list application allows the user to add notes to a list of items to be completed. This is done by typing in the note in the input field and pressing a button, which adds the note to the list. By clicking on the checkboxes next to each item added to the list, the user can indicated that the task has been completed. A screenshot of the to-do list application is provided in Figure 6 below:

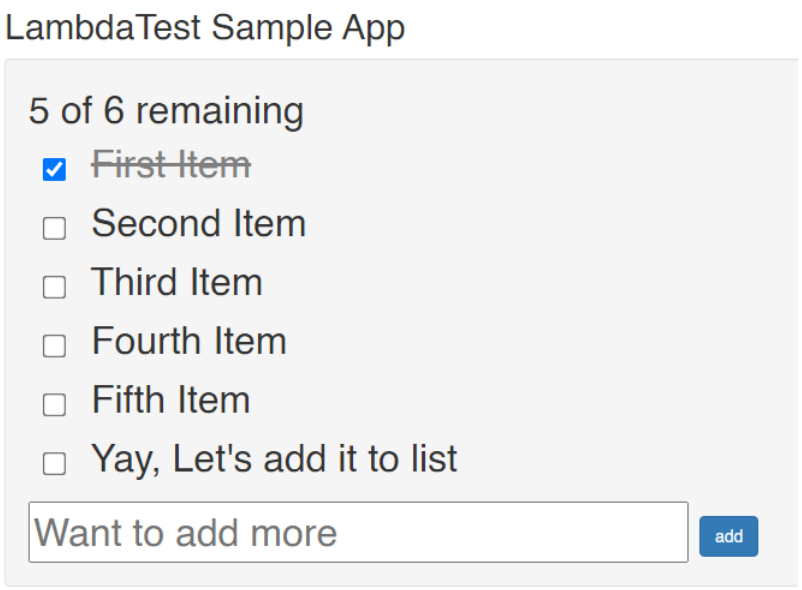

Figure 6: Screenshot of the to-do list application.

6.1 Motivation for the choice of acceptance testing tools to be compared

To limit the scope of the thesis, only three tools that can be used for BDD were chosen. The tools were chosen out of 15 testing tools that support BDD [40]. Due to financial limitations, as paid testing tools can be very expensive, only open-source tools were chosen. The tools were chosen based on their popularity, which was determined by the number of stars their projects have on the website GitHub.com, a code hosting platform. The amount of stars and forks on GitHub often reflects the popularity of the software. While these metrics depend on the age of the tools, this is not expected to be a major factor, as development of all of the tools started over 10 years ago, between 2005-2008 [41], [42], [43].

The first tool, Cucumber, was chosen because it is a well-recognized open-source testing tool that supports BDD [44]. Cucumber has 3200 stars on GitHub and 634 forks [45]. The Python port of Cucumber, called Behave, was chosen because of Python's concise syntax. Robot Framework was chosen as the second tool because it is popular and it also supports BDD. Robot Framework has 5600 stars on GitHub and 1600 forks [45], [46]. The last tool chosen was Concordion, which also supports BDD. Concordion is less popular, as it only has 199 stars on GitHub and 63 forks [47].

6.2 Selenium WebDriver

What all the selected tools have in common is that they allow to write high level test scenarios in their specification language, but on the lower level, they need to interact with the SUT. For this thesis, Selenium WebDriver [48] was chosen as the tool that interacts with the UI of the web application.

Selenium is probably the most popular test automation tool that can be used to automate a web browser to interact with a web application [3, p. 51-52]. Selenium is open-source, available in many programming languages, and supports the most common browsers [3, p. 56-57].

To perform actions on the web page, such as clicking buttons, or typing in text in an input field using Selenium, the HTML-elements will need to be accessed. This can be done using Selenium's built-in functions that can access elements on the page using their element type, attributes, or a combination of both. Below is an example of code written in Java that uses Selenium's library to select an element on the page of the type "input" with the class "inputLogin", and then inputting the text "username" into the input field.

```
WebElement textInput =
test.getBrowser().getDriver().findElement(By.xpath("//input[@class='inputLogin']"));

textInput.sendKeys("username");
```

6.2 Cucumber (Behave)

Cucumber is a testing tool that supports BDD. Cucumber supports the most common programming languages. With Cucumber it is possible to write acceptance tests in a language that is easy to understand for everybody in the team. Cucumber supports over 40 different spoken languages [49].

When using BDD with Cucumber, the product owners write scenarios, also known as user stories, that explain how the system should work from the end-user's perspective. There are three layers to Cucumber: the Gherkin layer, the "Glue" layer and the helper functions layer.

6.2.1 Gherkin layer

The first layer, Gherkin, is used to describe the feature in an easily understandable way, using Give-When-Then steps. The feature is described in a feature file and it is the first file to be created. A feature file contains the name and a description of the feature at the top, followed by one or several scenarios consisting of any number of Given-steps, one When-step and one or multiple Then-steps. The Given-steps might be left out if the system shares the set up inside the code, or if set up is not needed before starting the application. The When and the Then-steps, however, are mandatory. [50] This is the feature description for adding an item to a to-do list using Gherkin:

```
Feature: Test to add item to TODO list

    As a user
    I want to add items to a TODO list
    So that I can keep track of tasks to be done

    Scenario: Test TODO App
        Given I go to the TODO app to add an item
        When I enter an item to be added
        When I click on the add button
        Then I should verify the added item exists
```

6.2.2 Glue-layer

The second layer is the "Glue" where all of these steps are hooked up to some program code. In Cucumber language, these hooks are called *step definitions* and how they are implemented is highly dependent on the programming language used. In Python, the Given-, When-, and Then-steps are defined as annotations, followed by a regular expression that matches the line in the feature file. Each Given-, When-, Then-step is associated with a function that finds elements and performs actions on the page by calling helper functions that exist in a separate file. The helper functions are abstracted away in a different file to keep the step files small and to make it possible to re-use the helper functions in multiple step files. Below is the example of the Glue code for adding an item to a to-do list:

```
@given('I go to TODO app to add item')
def step(context):
context.helperfunc.open('https://lambdatest.github.io/sample-todo-app/')
    context.helperfunc.maximize()

@when('I enter item to add')
def enter_item_name(context):
    context.helperfunc.find_by_id('sampletodotext').send_keys(
        "Yay, Let's add it to the list")

@when('I click add button')
def click_on_add_button(context):
    context.helperfunc.find_by_id('addbutton').click()
```

```
@then('I should verify the added item')
def verify_todo_item(context):
    added_item_text = context.helperfunc.find_by_xpath(
        "//span[@class='done-false']").text
    assert added_item_text == "Yay, Let's add it to the list"
```

6.2.3 Helper functions layer

The last layer is the code that performs the actions, such as controlling the web driver, or making API calls. It is recommended to put this code in helper functions in a separate file to make the Glue-code easier to read [51]. The web driver is used to simulate user interactions on the page. Below are some examples of helper functions that utilize Selenium's built-in functions, using the to-do list application example. The functions are used to find elements on the page by their name and by their ID.

```
def find_by_name(self, name):
    return self._driver_wait.until(EC.visibility_of_element_located((By.NAME, name)))

def find_by_id(self, id):
    return self._driver_wait.until(EC.visibility_of_element_located((By.ID, id)))
```

6.3 Robot Framework

Robot Framework was initially developed as a generic test automation solution for Nokia Networks in 2005. The first open-source version of Robot Framework was released in 2008. [42] Robot Framework is used by many large companies, such as Finnair, Wärtsilä, and the Finnish tax administration. Robot Framework is a versatile tool that can be used for automated acceptance testing, as well as robotic process automation, among other things. Support for BDD using Gherkin is built into Robot Framework. Robot Framework is open-source and operating system and application-independent. The core framework runs on Python, JVM, and .NET. Many test libraries and tools have been developed that extend Robot Framework's capabilities. [52]

Acceptance tests with Robot Framework are written using keywords. This means that one does not have to write e.g. the code that controls the web driver, as that is handled by the tool. Instead, short keywords, such as "Title Should Be", "Click

Element", and "Open Browser" are used to control the web driver. It is recommended that the keywords are written to be reusable in many test cases to minimize code duplication. The two most important files when writing tests using Robot Framework are: the file that contains the test cases, and the file that controls the web driver.

6.3.1 Test case file

This file defines the keywords for the test cases. The settings for this file are defined at the top of the file. The Resource file that contains the keywords which control the web driver must be referenced in the settings. Below is a simple example of settings that contain the documentation, a reference to the Resource file, and keywords to be executed when the test is completed:

```
*** Settings ***
Documentation     A test suite with a single Gherkin style test.
Resource          resource.robot
Test Teardown     Close Browser
```

The test cases are specified below the settings. Each test case has unique and descriptive keywords. Below is an example of keywords for a test case using Robot Framework that opens up a browser, navigates to the URL of a to-do list application, clicks on checkboxes on the to-do list, adds a new item to the list, and verifies the item was added:

```
*** Test Cases ***
Add TODO item
    Given I go to the TODO app
    Then I click on the first checkbox and second checkbox
    When I enter the item "Yay! Added task to the list." to be added
    When I click the add button
    Then I should verify the item was added
```

The keywords are defined under the test cases. The description of the keywords must match the exact words used in the test cases. In the example below, "I go to the TODO app" is the description, and "Open browser to TODO app" is the keyword.

The actions of the keywords are defined in the Resources file using the same words as the keyword. The actions in the Resources file are used to control the web driver that performs actions on the page. Multiple keywords can be defined for each description. Below is an example of a keyword:

```
*** Keywords ***
I go to the TODO app
    Open browser to TODO app
```

6.3.2 Resource file

This file specifies what actions the web driver should take for each keyword mentioned in the test case file. Each top level keyword should be defined in the test case file. This file also contains settings at the top of the file. Below is an example of settings containing documentation and that specifies libraries that are used:

```
*** Settings ***
Documentation  A resource file with reusable keywords and variables.
...
...            The system specific keywords created here form our own
...            domain specific language. They utilize keywords
...            provided by the imported SeleniumLibrary.
...            by the imported SeleniumLibrary.
Library        SeleniumLibrary
```

Below the settings, the variables that are used in the file are defined. Variables are specified using a dollar sign and brackets syntax. Below is an example of some variables:

```
*** Variables ***
${SERVER}       lambdatest.github.io
${BROWSER}      Chrome
${TODO URL}     http://${SERVER}/sample-todo-app/
```

The keywords are defined below the variables. Below is an example of a keyword that controls the web driver using two keywords and that also defines another keyword (“TODO App Should Be Open”):

```
*** Keywords ***
Open Browser To TODO app
    Open Browser    ${TODO URL}    ${BROWSER}
    Maximize Browser Window
    TODO App Should Be Open
```

The “TODO App Should Be Open” keyword further maps to the “Title Should Be” keyword, which checks that the title of the web page is correct using Selenium WebDriver:

```
TODO App Should Be Open
    Title Should Be    Sample page - lambdatest.com
```

6.3.2.1 How the keywords map to SeleniumLibrary

The keywords used in the Resources file map to methods in SeleniumLibrary, the web testing library of Robot Framework. SeleniumLibrary utilizes the Selenium tool internally. [53] The keywords used to control the web driver must be identical to the names of the methods in the library, except with spaces instead of underscores, and capitalized words instead of lowercase words [54]. For instance, the keyword “Maximize Browser Window” maps to the method *maximize_browser_window* in the class *WindowKeywords* in the file *window.py* in SeleniumLibrary [53]. The code for the method is provided below:

```
@keyword
def maximize_browser_window(self):
    """Maximizes current browser window."""
    self.driver.maximize_window()
```

As another example, the “Title Should Be” keyword maps to the method title_should_be in the class BrowserManagementKeywords in the file

browsermanagement.py in SeleniumLibrary [53]. The code for the method is provided below:

```
def title_should_be(self, title: str, message: Optional[str] = None):
        """Verifies that the current page title equals ``title``.
        The ``message`` argument can be used to override the default error
        message.
        ``message`` argument is new in SeleniumLibrary 3.1.
        """
        actual = self.get_title()
        if actual != title:
            if message is None:
                message = f"Title should have been '{title}' but was '{actual}'."
            raise AssertionError(message)
        self.info(f"Page title is '{title}'.")
```

6.4 Concordion

Concordion is an open-source testing framework for Java-based projects that supports BDD. Concordion can turn requirements written in plain English into automated tests, similarly to Cucumber and Robot Framework. [47] Concordion supports writing specifications in HTML or Markdown. While Concordion only supports Java, it has been ported to Python (PyConcordion), C# (Concordion.NET), and Ruby (Ruby-Concordion).

6.4.1 Specification file

With Concordion, one starts by creating the specification first in Markdown or HTML. See Figure 7 below for a preview of the to-do list app's specification for the feature of adding an item to the list, written in Markdown. Concordion does not require that specifications are written in a certain format. It is possible to start by writing the specifications in the Given-When-Then format, and later change it to a more natural language, once you become familiar with thinking about the context, action and outcome of an example. [55]

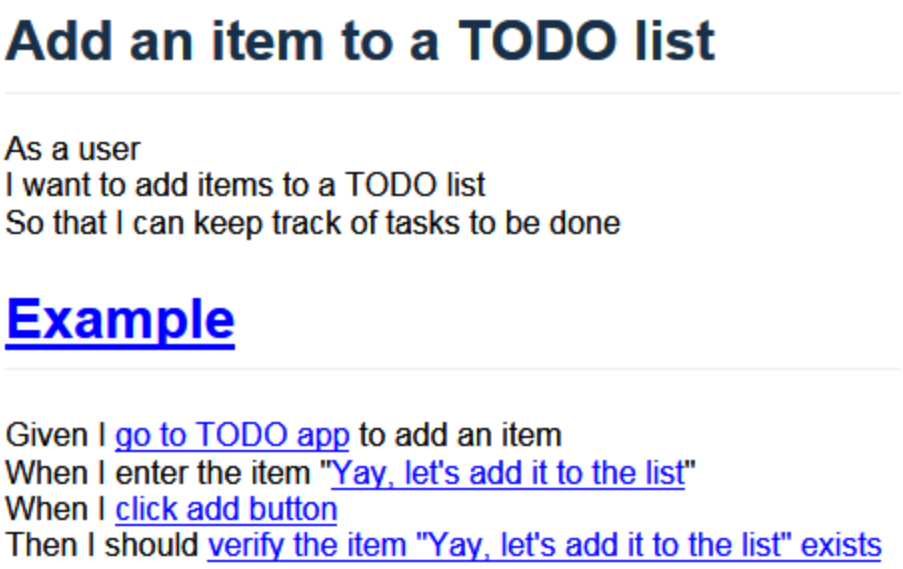

Figure 7: Preview of a Concordion specification written in Markdown.

The main difference between Concordion and the other tools chosen is that it is developed specifically for creating living documentation written in HTML or Markdown files. Living documentation can be accessed and read by all members of the team and it is directly tied to how the system works. The specifications can be written in normal language, and media files can also be added to the specifications.

To create executable specifications using Markdown documents and Concordion, commands need to be added to the specification. The commands have to be added as links in the Markdown document. A hyphen has to be added, to differentiate the Concordion commands from regular Markdown links. [56] In the specifications, you can specify variables from the text, run methods that exist in a Fixture file, do assertions, etc.

In the specification for adding an item to the list of the to-do list application, which is provided below in Markdown, the first method that is called in the Fixture file is *openTodoApp*. On the second line, the variable newItem is defined and given the value “Yay, added item to list”. The method *enterItem* is then called with *newItem* as the argument. Finally, the method *verifyItemExists* is called with the argument *newItem*, to check that the item has been added to the list.

```
# Test to add item to TODO list

As a user
I want to add items to a TODO list
So that I can keep track of tasks to be done

## [Example](-)

Given I [go to TODO app](- "openTodoApp()")
When I enter the item "[Yay, added item to list](- "#newItem")" [ ](-
"enterItem(#newItem)")
Then I should [verify the item exists](- "verifyItemExists(#newItem)")
```

6.4.2 Fixture file

The methods that are called from the specification are placed in the Fixture file. The function of the Fixture file is to open up different pages in the application and to run methods that exist on the pages that test different features on the pages. In the Fixture file (which is provided below) of the to-do list app, the browser, that is initialized in the method *openTodoApp,* is stored in a variable *browserInstance*, so that it can be used in the other methods that are called from the specification. The URL of the to-do list app is fetched from the *AppConfig* file.

```
public class TodoAddItemFixture extends CubanoDemoBrowserFixture {

    private BrowserBasedTest browserInstance;

    public String getTodoAppURL() {
        return AppConfig.getInstance().getTodoAppUrl();
    }
    public void openTodoApp() throws Exception {
        browserInstance = TodoAppPage.open(this);
    }
    public void enterItem(String newItem) throws Exception {
        TodoAppPage.enterItem(browserInstance, newItem);
    }
    public void addItem(String newItem) throws Exception {
```

```
        TodoAppPage.addItem(browserInstance);
    }
    public void verifyItemExists(String item) throws Exception {
TodoAppPage.verifyItemExists(browserInstance, item);
    }
}
```

6.4.3 Page Object file

In the Page Object files, the code for the web driver is implemented, if a web driver such as Selenium is used to test the web application. Each page should have a Page Object class. This is to make sure not all test cases are put into one giant Page Object. In the to-do list app's Page Object provided below, the *enterItem* method enters text into an input field using Selenium WebDriver, while the *addItem* method clicks on a button which adds the item to the list, also using Selenium WebDriver. The open method opens the browser and navigates to the to-do list app's URL. The *verifyItemExists* method looks for a certain element on the page and checks whether it contains the new item.

```
public class TodoAppPage extends PageObject<ConcordionEvernoteLoginPage> {

    public TodoAppPage(BrowserBasedTest test) throws InterruptedException {
        super(test);
    }

public static BrowserBasedTest enterItem(BrowserBasedTest test, String newItem)
throws InterruptedException {
        WebElement newItemInput =
test.getBrowser().getDriver().findElement(By.id("sampletodotext"));
        newItemInput.sendKeys(newItem);
        return test;
    }

    public static BrowserBasedTest addItem(BrowserBasedTest test) throws
InterruptedException {
        WebElement addItemButton =
test.getBrowser().getDriver().findElement(By.id("addbutton"));
        addItemButton.click();
        return test;
    }

    public static void verifyItemExists(BrowserBasedTest test, String correctItem)
```

```
throws InterruptedException {
        WebElement newItemElement =
test.getBrowser().getDriver().findElement(By.xpath("//span[@class='done-false"));
        String newItemElementText = newItemElement.getText();
        Assert.assertThat(newItemElementText,
CoreMatchers.containsString(correctItem));
    }

    public static BrowserBasedTest open(BrowserBasedTest test) throws
InterruptedException {

test.getBrowser().getDriver().navigate().to(AppConfig.getInstance().getTodoAppUrl()
);

        return test;
    }
}
```

7. Evaluation of the selected testing tools via a reference application

The first sections of this chapter explains the methods, and the motivation behind choosing the methods used to evaluate the testing tools. To better evaluate the tools, a longer test case is implemented on a popular web application in the later sections of this chapter. In the later sections, the implementation of the test case is also explained in detail for each tool.

7.1 Methods used to evaluate the testing tools

To compare the testing tools, it must be known what priorities testers have when it comes to selecting an acceptance testing tool. Surveys and literature were examined to discover the top priorities testers have when selecting a test automation tool. These priorities are used in Chapter 8 to compare the testing tools, to decide which is best suited for BDD in web applications. Two surveys of testers [57], [58] and three pieces of literature, [59], [60], [61] were examined to find important priorities. Omitting licensing and support costs, as all tools used are free, the five important priorities identified were:

- Ease of use
- Good test reports
- Popularity. training, documentation, and tutorials
- Ease of adoption and configuration
- Efficiency

To evaluate the ease of use, ease of adoption, and efficiency of the tools, a test case will be implemented and run on a web application using each tool. The documentation available on the websites of each tool and online sources will be used to evaluate the test reporting capabilities, as well as the popularity, training, documentation, and tutorials available. The tools will be evaluated based on 10 questions in the five categories. The questions are provided below in Table 1:

Table 1: Evaluation questions

Criteria	ID	Questions
Usability	Q1	What is the SUS score?
	Q2	Is it possible to write tests without coding?
	Q3	How many lines of code is needed to implement the test case?
Test reporting	Q4	Can the tool generate highly configurable test reports automatically, and if not, is it possible to integrate a test reporting tool?
Popularity, training resources and community support	Q5	How many visitors does the tool's website get from search engine results pages per month?
	Q6	How many times is the tool searched on Google per month?
	Q7	Are there workshops, paid or free training, certifications or conventions?
	Q8	Is there community support, and if so, on what platform?
Ease of adoption	Q9	Can a demo project from the documentation be installed and executed within 60 minutes?
Efficiency	Q10	What was the execution time for the test case?

7.1.1 Evaluating usability

Three factors will be used to evaluate the usability of the tools: the System Usability Score (SUS), whether coding is required to implement test cases using the tool, and how many lines of code (LOC) are needed to implement the test case in the later sections of this chapter.

The SUS system is a reliable way to quickly evaluate the usability of a system, and it has been in use for over 30 years [62]. SUS has also been used in previous research to measure the usability of automatic testing tools [61]. The SUS system uses a

Likert scale [63] that includes 10 questions that the users have to answer with a number. The 10 questions used in SUS are:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The answers are given in a numerical form, from 1-5, 1 meaning that the participant strongly disagrees, and 5 meaning that the participant strongly agrees with the statement. Calculations are then done to transform the numerical answers into a value ranging from 0 to 100, 0 being the least user friendly, and 100 being the most user friendly.

As usability is highly subjective, it is commonly measured using many participants to get a more objective result. However, due to a lack of access to participants who were able to test the tools and answer questions, the 10 questions will only be answered by the author of this thesis. A low number of participants makes the SUS scores more subjective, as e.g. the programming knowledge of the individual will affect the answers greatly. For instance, as the author has intermediate programming knowledge, the answers may not reflect the opinions of users with very little programming knowledge, as they may find it harder to use the tools. Inversely, individuals with more extensive programming knowledge may find the tools easier to use. Despite this, SUS is useful in this thesis because the scores indicate the relative usability of the tools.

Because one of the goals of BDD is to engage all members of the team, even those that lack coding knowledge, it is important to consider whether it is possible to write test cases using the tools without coding. Also, if the tool does not require coding, testers with less, or no coding knowledge can be hired and trained quicker, saving resources. Tools that do not require coding are expected to be easier to use for people without coding experience.

The number of LOC required to implement the same test case using the different tools is also compared. If fewer LOC are required, the tool is determined to be easier to use. This is especially true for users with limited coding skills. Fewer LOC also means that the tests can be written quicker and that there likely will be fewer errors.

7.1.2 Evaluating test reporting

Test reporting is important, as it provides information about how the test execution went, and e.g. helps with debugging the application. Three factors will be used to evaluate the test reporting capabilities of each tool: whether the reports are automatically generated, whether the reports are highly configurable, and whether it is possible to integrate a test reporting tool if the tool does not have built-in test reporting.

7.1.3 Evaluating popularity, training resources and community support

Four factors will be used to evaluate the popularity, training resources, and community support of the tools. The factors are: how many website visitors the tool's website receives per month, how many Google searches are performed for the tool per month, whether there is paid training, certifications, conventions and workshops, and whether there is community support available.

To evaluate the popularity of the tools, SERanking [64], a tool that is used to analyze search engine traffic will be used. The tool will be used to determine approximately how much traffic the testing tools' websites receive from Google's search engine results pages per month [65], and how many times the tool is searched per month [66]. If the name of the testing tool is ambiguous, as is the case with Cucumber, the

number of searches that contain the tool's name and terms related to testing will be summed to get the overall search volume.

Each tool's website will also be examined to determine what training resources are available, and whether community support is provided, such as forums and Slack channels.

7.1.4 Evaluating ease of adoption and configuration

The ease of adoption and configuration will be evaluated based on how quickly the tool can be installed, configured, and a basic demo project from a tutorial on the tool's website can be run. If this can be accomplished within 60 minutes, the tool is considered to be easy to install and configure.

7.1.5 Evaluating efficiency

To evaluate the efficiency, the execution times for the same test case will be compared for each tool.

7.2 Web application to be tested

The web application that was chosen to be tested was Evernote [67], a popular note-taking web application. Evernote allows the user to store and organize notes. Evernote was chosen because it is a popular and relatively simple web application, which makes it a good choice for writing short test cases that show the features of, and how to implement simple acceptance tests using the testing tools.

The tools will be used to test the process of logging in and adding a note using Evernote. The test will check that the note has been added by checking that the title of the latest note is correct. The same test case will be written using Cucumber, Robot Framework and Concordion.

7.3 Implementing and running the tests with Cucumber

For Cucumber, the Python port of Cucumber called Behave was used. The feature description for adding a note in Evernote in the *AddANote.feature* file is provided below:

```
Feature: Add a note in Evernote

    As a user
    I want to add a note in Evernote
    So that I can access it later

    Scenario: Add a note in Evernote
        Given I go to the Evernote app
        When I write a note
        Then I should verify the added note
```

As described previously, the “Given”, “When”, “Then” keywords hook into the methods in the steps file using annotations which mention the keywords. The code in the steps file is provided below:

```
@given('I go to Evernote to add a note')
def start(context):
    context.helperfunc.open('https://www.evernote.com/Login.action')
    context.helperfunc.maximize()
    context.helperfunc.find_by_id('username').send_keys(
        "username")
    context.helperfunc.find_by_id('loginButton').click()
    context.helperfunc.find_by_id(
        'password').send_keys("password")
    context.helperfunc.find_by_id('loginButton').click()
```

This code visits the Evernote login page, finds the username input field and inputs a username, and finds the login button by its ID on the page, and clicks it. Then the password field appears; it is selected by its ID, and the password is entered, after which the login button is pressed and the user is logged in.

```
@when('I write a note')
def add_note(context):
    # click add note
    context.helperfunc.find_by_id('qa-CREATE_NOTE').click()
```

```
    time.sleep(2)
    # switch to note editor iframe
    context.helperfunc.switch_frame("qa-COMMON_EDITOR_IFRAME")
    # type in title
    context.helperfunc.find_by_xpath(
        "//textarea[@class='_3wdDa _2gdsJ']").send_keys("Test title")
    # type in note
    context.helperfunc.find_by_xpath("//div[@class='para']").send_keys(
        "Test text")
    # switch back to parent document
    context.helperfunc.switch_back()
```

This code finds the button for creating a new note and clicks it. Then it waits for two seconds because it takes some time for the iframe which is used for editing the note to load. Then it switches to the editor iframe, using the iframe's ID on the page. Then it finds the textarea that is used for inputting the title of the note, and it inputs the title "Test title", using the send_keys function. Then it does the same thing for the body of the note. Finally, it switches back to the main document from the editor iframe.

```
@then('I should verify the added note')
def verify_note_exists(context):
    time.sleep(2)
    note_title_text = context.helperfunc.find_by_xpath(
        "//div[@class='OzCyOGOuSNLMCuvVYJesK Y8p50x11YFkD12NFEwqvc
_2tWVs2YDKEIUbIZYp_SNA6']/span").text

    assert note_title_text == "Test title"
```

The last step is to verify that the note was added. This is done by finding the element that contains the title of the latest note. The title is located within a span element inside a div element with a rather long class name. Then it asserts that the title text is "Test title".

The code for the helper function, which uses Selenium WebDriver's Python API to control the web driver is provided below:

```
class HelperFunc(object):
    __TIMEOUT = 30

    def __init__(self, driver):
        super(HelperFunc, self).__init__()
        self._driver_wait = WebDriverWait(driver, HelperFunc.__TIMEOUT)
        self._driver = driver

    def open(self, url):
        self._driver.get(url)

    def maximize(self):
        self._driver.maximize_window()

    def close(self):
        self._driver.quit()

    # Helper functions that are used to identify the web locators in the Selenium
Python tutorial

    def find_by_xpath(self, xpath):
        return self._driver_wait.until(EC.visibility_of_element_located((By.XPATH,
xpath)))

    def find_by_name(self, name):
        return self._driver_wait.until(EC.visibility_of_element_located((By.NAME,
name)))

    def find_by_id(self, id):
        return self._driver_wait.until(EC.visibility_of_element_located((By.ID,
id)))

    def find_by_class(self, className):
        return
self._driver_wait.until(EC.visibility_of_element_located((By.CLASS_NAME,
className)))

    def switch_frame(self, id):
```

```
        return self._driver.switch_to.frame(id)

    def switch_back(self):
        self._driver.switch_to_default_content()
```

The test case was run using the following command:

```
behave -f allure_behave.formatter:AllureFormatter -o /allure-reports
features/AddANote.feature
```

Allure [68], a test report tool that can be easily integrated with Behave, was used to generate the test reports because Behave only supports creating test reports in JSON or XML format, neither of which are easily readable [69]. In the command, it was specified that only the *AddANote* feature should be run. Running the test case using Behave took 24.070 seconds, as can be see in the report created by Allure in Figure 8 in Chapter 8. In total, 100 LOC were used for the Gherkin feature file, the steps file, and the helper functions file. For the Gherkin file, 10 LOC were used. For the steps file, 49 LOC were used. For the helper file, 41 LOC were used.

7.4 Implementing and running the test with Robot Framework

The Gherkin file for the test case implemented using Robot Framework is provided below. When the test is run, Robot Framework looks for the keywords in the resource.robot file in order, starting with the keyword “Open browser to Evernote”, and ending with “Verify added note”, which takes the variable *${correct_note_title}*.

```
*** Settings ***
Documentation     A sample test case for adding a note with Evernote
...
...               This test checks the feature of adding a note.
Resource          resource.robot
Test Teardown     Close Browser

*** Test Cases ***
Add a note
```

```
    Given I go to the Evernote app
    When I write a note with title "Test title" and text "Test text"
    Then I should verify the added note exists and has the title "Test title"

*** Keywords ***
I go to the Evernote app
    Open browser to Evernote
    Login to Evernote

I write a note with title "${note_title}" and text "${note_text}"
    Write a note    ${note_title}    ${note_text}

I should verify the added note exists and has the title "${correct_note_title}"
    Verify added note   ${correct_note_title}
```

Selenium WebDriver is then controlled in the resource.robot file to perform actions on the page using keywords, such as “Open Browser”, and “Input Text”. At the last step, an assertion is performed using the keyword “Element Text Should Be”, to check that the latest note has the correct title. The *resource.robot* file is provided below:

```
*** Settings ***
Documentation     A sample test case for adding a note with Evernote
Library           SeleniumLibrary

*** Variables ***
${SERVER}          evernote.com
${BROWSER}         Chrome
${DELAY}           0
${VALID USER}      username
${VALID PASSWORD}  password
${LOGIN URL}       https://${SERVER}/Login.action

*** Keywords ***
Open Browser To Evernote
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}
```

```
Login To Evernote
    Input Text  username    ${VALID USER}
    Click Button    loginButton
    Wait Until Element Is Visible   password
    Input Password  password    ${VALID PASSWORD}
    Click Button    loginButton

Write A Note
    [Arguments]    ${note_title}    ${note_text}
    Wait Until Element Is Visible   qa-CREATE_NOTE  timeout=15
    Click Button    qa-CREATE_NOTE
    Wait Until Element Is Visible   qa-COMMON_EDITOR_IFRAME
    Select Frame    qa-COMMON_EDITOR_IFRAME
    Input Text    //textarea[@class='_3wdDa _2gdsJ']     ${note_title}
    Input Text    //div[@class='para']    ${note_text}
    Unselect Frame

Verify Added Note
    [Arguments]    ${correct_note_title}
    Sleep  2s
    ${added_note_title}   Get WebElement    //div[@class='OzCyOGOuSNLMCuvVYJesK
Y8p50x11YFkD12NFEwqvc _2tWVs2YDKEIUbIZYp_SNA6']/span
    Element Text Should Be   ${added_note_title}     ${correct_note_title}
```

The test case was run with the following command:

```
robot evernote_tests/gherkin_add_note.robot
```

According to the built in test reporting tool of Robot Framework, running the test case took 27.144 seconds. In total 67 LOC were used for the Gherkin and the resource.robot files. When running the test, an output file in XML format, and a log and report file in HTML format was created. See Figure 9 in Chapter 8 for a screenshot of the generated report.

7.5 Implementing and running the tests with Concordion

The specification for the Concordion test is provided below in Markdown format. The specification contains the commands which run the methods in the Fixture.

```
# Adding a note in Evernote

As a user
I want to add a note in Evernote
So that I can access it later

## [Example](-)

Given I [go to the Evernote app](- "loginToEvernote()")
When I write a note with the title "[Test title](- "#noteTitle")" [ ](-
"writeANote(#noteTitle)")
Then I should [verify the added note exists and has the title "Test title"](-
"verifyNoteWithTitleExists(#noteTitle)")
```

The first method that is called in the Fixture is *loginToEvernote*, which starts the browser, navigates to Evernote and logs in. On the second line, the variable *noteTitle* is defined and given the value "Test title". The method *writeANote* is then called with *noteTitle* as the argument. Finally, the method *verifyNoteWithTitleExists* is called with the argument *noteTitle*. The Fixture class that implements the methods is provided below:

```
public class ConcordionAddNoteFixture extends CubanoDemoBrowserFixture {

    private BrowserBasedTest browserInstance;

    public String getConcordionEvernoteLoginURL() {
        return AppConfig.getInstance().getConcordionEvernoteLoginUrl();
    }
    public void loginToEvernote() throws Exception {
        browserInstance = ConcordionEvernoteLoginPage.open(this);
        ConcordionEvernoteLoginPage.loginToEvernote(browserInstance);
    }
    public void writeANote(String noteTitle) throws Exception {
        ConcordionEvernoteAppPage.writeANote(browserInstance, noteTitle);
    }
    public void verifyNoteWithTitleExists(String correctNoteTitle) throws Exception
{           ConcordionEvernoteAppPage.verifyNoteWithTitleExists(browserInstance,
correctNoteTitle);
    }
```

```
}
```

The browser that is initialized in the method *loginToEvernote* is stored in a variable *browserInstance*, so that it can be used in the other methods that are called from the specification. The URL of the login page is fetched from the *AppConfig* file. The *loginToEvernote* method is called from the *ConcordionEvernoteLoginPage* PageObject class, while the *writeANote* and *verifyNoteWithTitleExists* methods are called from the *ConcordionEvernoteAppPage* PageObject class.

Below is the code for the Evernote login page PageObject class, which contains the method for logging in a user:

```
public class ConcordionEvernoteLoginPage extends
PageObject<ConcordionEvernoteLoginPage> {

    public ConcordionEvernoteLoginPage(BrowserBasedTest test) throws
InterruptedException {
        super(test);
    }

    public static BrowserBasedTest loginToEvernote(BrowserBasedTest test) throws
InterruptedException {
        WebElement usernameInput =
test.getBrowser().getDriver().findElement(By.id("username"));
        usernameInput.sendKeys("username");

        WebElement loginButton =
test.getBrowser().getDriver().findElement(By.id("loginButton"));
        loginButton.click();

        WebDriverWait wait = new WebDriverWait(test.getBrowser().getDriver(), 10);
        WebElement passwordInputElement = wait.until(
                ExpectedConditions.visibilityOfElementLocated(By.id("password")));
        passwordInputElement.sendKeys("password");

        loginButton.click();
        return test;
    }
```

```
    public static BrowserBasedTest open(BrowserBasedTest test) throws
InterruptedException {

test.getBrowser().getDriver().navigate().to(AppConfig.getInstance().getConcordionEv
ernoteLoginUrl());

        return test;
    }
}
```

Below is the code for the Evernote app page PageObject class. This page is the main page for the web application, where notes can be viewed and added.

```
public class ConcordionEvernoteAppPage extends
PageObject<ConcordionEvernoteAppPage> {

    public ConcordionEvernoteAppPage(BrowserBasedTest test) throws
InterruptedException {
        super(test);
    }

    public static BrowserBasedTest writeANote(BrowserBasedTest test, String
noteTitle) throws InterruptedException {

        WebDriverWait wait = new WebDriverWait(test.getBrowser().getDriver(), 10);
        WebElement addNoteButton = wait.until(

ExpectedConditions.visibilityOfElementLocated(By.id("qa-CREATE_NOTE")));
        addNoteButton.click();

        WebElement noteEditorFrame = wait.until(

ExpectedConditions.visibilityOfElementLocated(By.id("qa-COMMON_EDITOR_IFRAME")));
        test.getBrowser().getDriver().switchTo().frame(noteEditorFrame);

        WebElement noteTitleInput =
test.getBrowser().getDriver().findElement(By.xpath("//textarea[@class='_3wdDa
_2gdsJ']"));
        noteTitleInput.sendKeys(noteTitle);
```

```
        WebElement noteTextInput =
test.getBrowser().getDriver().findElement(By.xpath("//div[@class='para']"));
        noteTextInput.sendKeys("Test text");

        test.getBrowser().getDriver().switchTo().defaultContent();
        return test;
    }

    public static void verifyNoteWithTitleExists(BrowserBasedTest test, String
correctTitle) throws InterruptedException {
        Thread.sleep(2000);

        WebElement noteTitleElement =
test.getBrowser().getDriver().findElement(By.xpath("//div[@class='OzCyOGOuSNLMCuvVY
JesK Y8p50x11YFkD12NFEwqvc _2tWVs2YDKEIUbIZYp_SNA6']/span"));
        String noteTitleElementText = noteTitleElement.getText();
        Assert.assertThat(noteTitleElementText,
CoreMatchers.containsString(correctTitle));
    }
}
```

Implementing the test case took approximately 165 LOC. The test case was executed by running the specification file as a JUnit test. According to the built in test reporting tool of Concordion, running the test case took 27.368 seconds. The results of the test case were stored automatically in an HTML file. See Figure 10 in Chapter 8 for a screenshot of the generated test report.

8. Analysis

This chapter discusses the differences between the tools within the five categories that were discovered to be important to testers in Chapter 7. To make it easier to compare the differences, 10 questions (see Table 1), each belonging to one of the five categories, were answered for each tool. The results are summarised in Table 2 below. The following sections discuss the results in depth.

Table 2: Final testing tool assessment results

ID	**Tools**		
	Cucumber (Behave)	**Robot Framework**	**Concordion**
Q1	75	85	42.5
Q2	Yes	No	Yes
Q3	100 LOC	67 LOC	167 LOC
Q4	Possible to integrate a test reporting tool	Automatically generated, highly configurable test reports	Automatically generated test reports
Q5	57,000 visitors / mo.	9,600 visitors / mo.	33 visitors / mo.
Q6	6,500 searches / mo.	4,400 searches / mo.	20 searches / mo.
Q7	Free and paid courses, workshops	Free and paid training, workshops, and certifications	Free guides
Q8	Community forum, Slack channel, mailing list	Community forum, Slack channel	Google Group support
Q9	Yes	Yes	No
Q10	24.070 s	27.144 s	27.368 s

8.1 Ease of use

When comparing System Usability Scale (SUS) scores to determine the usability of the tools, it is useful to know what the industry standards are. A SUS score of 80.3 or higher is comparable to an “A” in the academic grading system in the United States, which uses letter grades ranging from “F” to “A”, with F being the lowest, and A+ being the highest grade. A SUS score of 80.3 or higher thus means that the usability is excellent, whereas a score of 68 is comparable to a C, meaning that there are improvements to be made. A SUS score of 51 or lower is the equivalent to an “F”, a failed grade, indicating that the usability of the system needs to be improved urgently. [62]

8.1.2 Cucumber (Behave)

Cucumber had the second-highest SUS score of 75. Detailed documentation and demo projects are available on the tool's website that can be used as a reference. Cucumber's website also provides many video tutorials that explain how to implement tests using the tool. Writing tests with Cucumber requires coding, however, a user that has completed an introductory course in programming is expected to be able to write tests using the provided documentation and learning resources. It was easier to get started with Cucumber compared to Robot Framework, as writing tests with Cucumber did not require learning how to use keywords. However, writing tests with Robot Framework is expected to be faster once the user understands how to use the keywords, as it does not require coding. Implementing the test with Cucumber required 100 LOC.

8.1.1 Robot Framework

The tool with the highest SUS score, 85, was Robot Framework. Writing tests using Robot Framework was easy, due to the good documentation and the availability of demo projects and tutorials on the tool's website. Using Robot Framework’s keywords required a learning curve, but the fact that no coding was needed makes it possible for users without coding knowledge to write tests using the tool. Once it became clear how the keywords should be used, writing tests using the tool was fast

and easy. Robot Framework required the least lines of code, 67, to implement the test case.

8.1.3 Concordion

Concordion had the lowest SUS score at 42.5, indicating that the usability needs to be improved. Implementing the test case was more difficult using Concordion, compared to the other tools. For instance, there were many errors in the Cubano demo project that were caused by missing libraries. It is unclear whether the errors were caused by user error due to lacking documentation, or whether the provided demo project was outdated.

The tutorials on Concordion's website were also less clear compared to the other tools. For instance, the architecture of the codebase was not explained in the tutorial of the demo project on Concordion's website, making it difficult to understand which files did what in the demo project. To use Selenium with Concordion a separate library, Cubano had to be installed. The demo project for Cubano was larger than Cucumber's and Robot Framework's demo projects. The demo project contained more than 40 Java files, and the function of most of the files were not explained in the documentation. This made it difficult to modify the demo project to implement the test case. A considerable amount of trial and error, as well as external sources, were used when implementing the tests. Implementing the test in Concordion also required the most lines of code, 167. However, the fact that the Java version of Concordion was used should be taken into consideration, as implementing the same feature in Java typically requires more lines of code than Python.

Concordion has one advantage, however, when it comes to usability. With Concordion, it is possible to write specifications in normal language using proper punctuation, paragraphs, etc., while the other tools require that the Gherkin language is used. [70]

8.2 Test reporting

8.2.1 Cucumber (Behave)

Using Behave, the Python port of Cucumber, it is possible to output JUnit-compatible reports without installing extra dependencies, by adding --junit at the end of the command when running the tests. However, this only produces reports in XML format that are difficult to read. To generate HTML reports that are easier to read, Allure, a test-reporting library that can integrate with Behave can be installed. [71]. See Figure 8 below for a a screenshot of the test report generated using Allure.

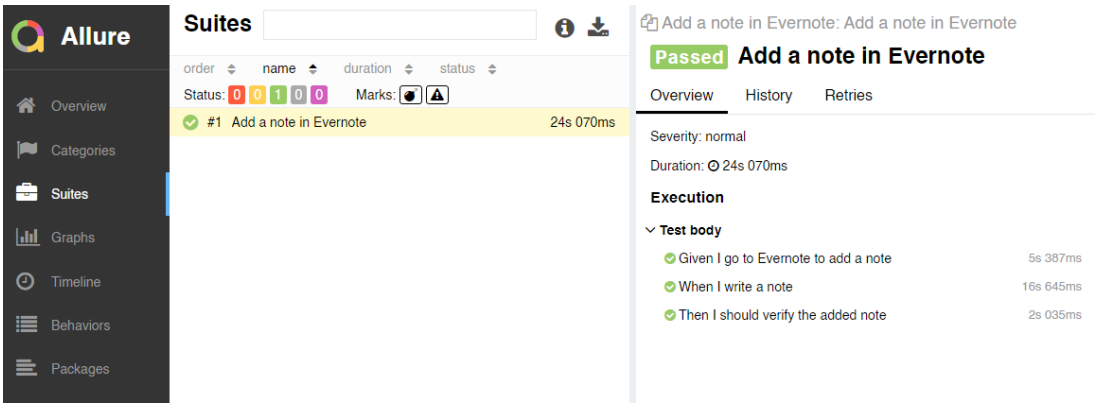

Figure 8: Screenshot of the Allure test report.

8.2.2 Robot Framework

During the test execution, Robot Framework generates XML output files. Rebot, a tool that is included with Robot Framework, automatically post-processes the XML output files. Rebot is able to automatically generate easily readable test reports in many formats, including HTML. Rebot can also be used separately to merge and combine results and to create custom logs and reports. [72] See Figure 9 below for a screenshot of the test report generated by Robot Framework.

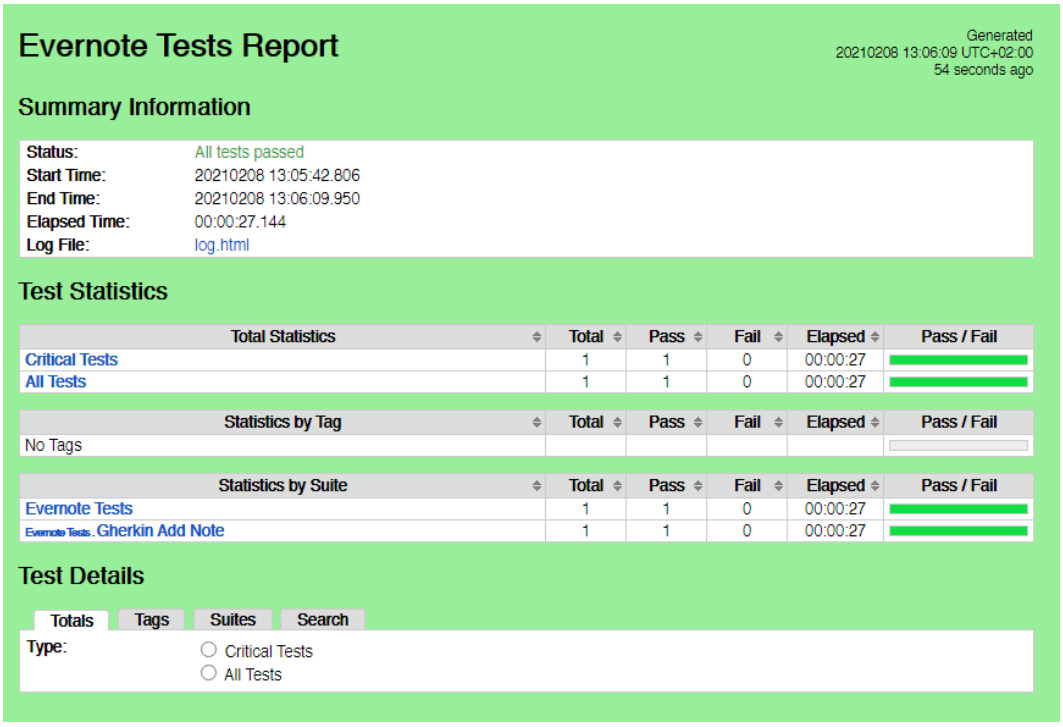

Figure 9: Test report generated by Robot Framework.

8.2.3 Concordion

Concordion automatically generates test reports in HTML format. No information about the configurability of the reports was found in Concordion's documentation. However, as the software is open-source, it is possible to modify the code to configure the test reports. See Figure 10 below for a screenshot of the test report generated by Concordion.

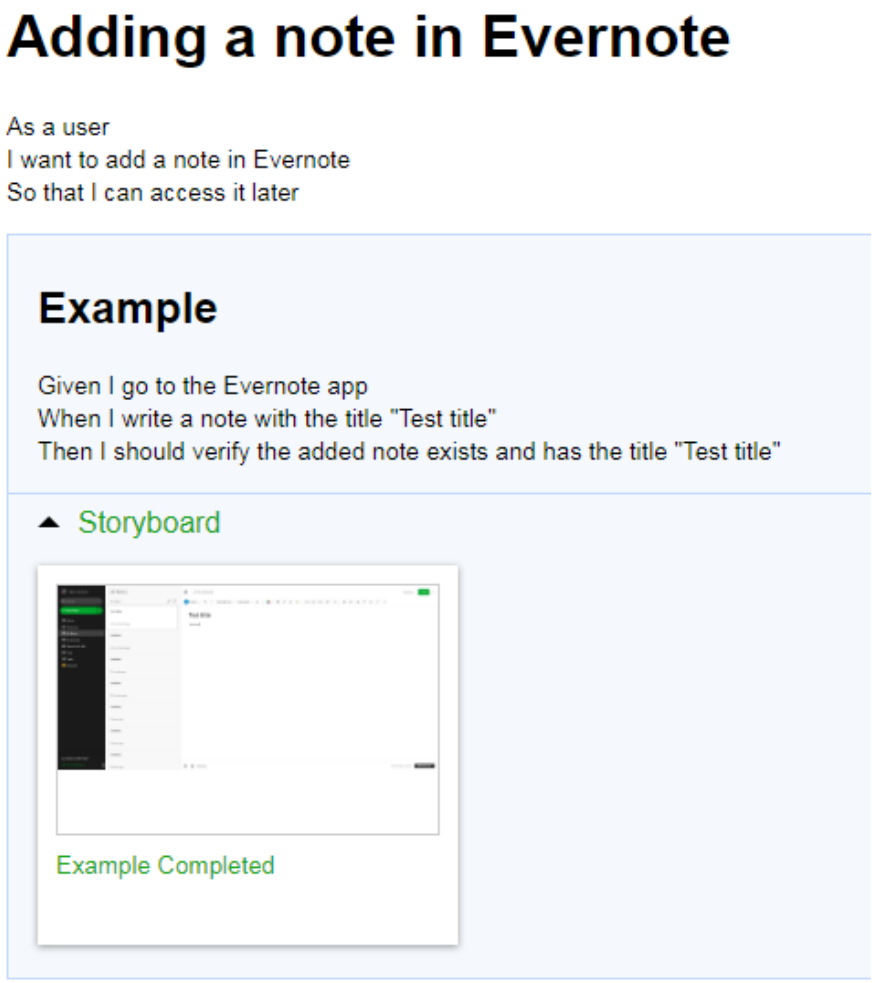

Figure 10: Concordion test output HTML file.

8.3 Available documentation, support, and popularity

8.3.1 Cucumber

Cucumber has substantial support and documentation. On Cucumber's website, there are free and paid courses on how to use Cucumber [73], as well as e-books, case studies, webinars, and events that explain how to use Cucumber and BDD [74]. There is also an open forum, community Slack channel and mailing list available for support [75], [76].

The Cucumber website receives approximately 57,100 visitors organically from Google's search engine results pages per month. The keywords "cucumber testing",

“cucumber test”, “cucumber testing tool” are searched a total of approximately 6500 times per month. It is therefore the most popular testing tool out of the three tools compared.

8.3.2 Robot Framework

There are many guides and tutorials available for Robot Framework [77]. The Robot Framework foundation also provides training and certifications and hosts yearly conferences [78]. There is also a forum available for Robot Framework users. The forum is active, with new posts being created daily. [79] There is also a Slack channel available for support.

Robot Framework’s website receives approximately 9,600 visitors organically from Google’s search engine results pages per month. The keyword “Robot Framework” is searched a total of approximately 4400 times per month.

8.3.3 Concordion

Concordion’s website provides documentation and a guide on how to get started with Concordion. There are no video tutorials or other learning resources available on the tool’s website. There is a Google Group available for Concordion, where the project leader answers questions [80].

Concordion’s website receives approximately 33 website visitors organically from Google’s search engine results pages per month. The keyword “Concordion” is searched a total of approximately 20 times per month.

8.4 Ease of adoption and configuration

Using official tutorials and demo projects, it was possible to install, configure, and run a simple project with both Cucumber and Robot Framework within 60 minutes. This process took many hours using Concordion, mostly because of import errors that had to be fixed, but also because of inadequate documentation.

8.5 Efficiency

Cucumber was approximately three seconds faster at running the test case compared to the other tools. The test took 24.070 seconds to run using Cucumber, 27.144 seconds to run using Robot Framework, and 27.368 seconds to run using Concordion. Cucumber was thus approximately 12% faster at running the test compared to the other tools. Multiple, longer test cases would have to be run to more accurately determine the differences in execution times.

9. Conclusion and future research

This thesis compared three testing tools that support BDD in web applications, based on five criteria testers find important when choosing a testing tool. The goal was to help testers choose the correct testing tool for BDD in web applications. The tools were evaluated by implementing and running a test case on a web application. Documentation, past research, and literature were also examined when evaluating the tools. The results of comparing the tools in the five categories were summarised in Table 2.

The results show that both Behave (the Python port of Cucumber) and Robot Framework are good choices for BDD in web applications, as they are easy to use, have good documentation and many available learning resources, and they are easy to adopt. Both Concordion and Cucumber require that the users have coding knowledge. Robot Framework, on the other hand, is also suitable for users that do not have coding experience, due to its keyword syntax.

If Concordion is used for BDD in web applications, it needs to be taken into account that the tool has lacking documentation, which makes the tool harder to adopt and use. Concordion is also less popular than the other tools, by a large margin. Concordion can be used if more elaborate specifications in HTML or Markdown format are needed. Concordion also has fewer learning resources available and less community support. For these reasons, Concordion is more suitable for testers with good coding knowledge and troubleshooting abilities.

To better determine the differences in efficiency between the tools in future research, a longer test case can be run for each tool. In this thesis, only one short test case was run, which makes the differences in execution times small, as the execution times were short. Of particular interest would be whether the differences in execution times between Cucumber and the other tools increases with a longer test case. To more objectively measure usability in future research using the SUS system, more participants are required. As only the author answered the SUS questionnaire, the SUS scores only indicate the usability of the tools relative to each other.

Swedish summary

Överblick över Behaviour-Driven Development-verktyg för webbapplikationer

Företag och privatpersoner har blivit allt mer beroende av mjukvara. På senare tid har speciellt webbapplikationer blivit populära för att lagra och organisera information, samarbeta inom företag, köpa och sälja varor och aktier, underhållning och mycket mer. Att bygga välfungerande mjukvara är ingen lätt uppgift. Nästan hälften av alla mjukvaruprojekt misslyckas på något sätt, och webbapplikationer är inget undantag. En vanlig orsak till detta är brister i kommunikationen mellan kunden och mjukvaruutvecklarna. Dålig kommunikation mellan kunden och utvecklarna leder till oklara eller felaktiga krav på mjukvaran. Detta förlänger mjukvaruutvecklingsprojektet och ökar risken för att mjukvaran inte är av nytta för kunden. En annan vanlig orsak till att mjukvaruprojekt misslyckas är otillräcklig testning. Detta leder ofta till buggig mjukvara som inte uppfyller kundens krav.

Det huvudsakliga målet med Behaviour-Driven Development (BDD), en relativt ny metod för att utveckla mjukvara, är att förbättra kommunikationen mellan intressenterna i mjukvaruutvecklingsprojektet. Genom att skapa användarberättelser tillsammans med kunden får man reda på vilka egenskaper mjukvaran bör ha, hur egenskaperna bör fungera ur användarens perspektiv och hur egenskaperna bör prioriteras. Många testverktyg som stöder BDD har utvecklats som kan användas för att översätta användarberättelser till automatiskt körbara acceptanstester. Acceptanstesternas mål är att försäkra att egenskaperna uppfyller kundens krav. Då man använder BDD fungerar användarberättelserna både som lättförståelig dokumentation av egenskaper och som test som försäkrar att egenskaperna uppfyller kundens krav.

I avhandlingen jämfördes tre testverktyg som kan användas för BDD och för att testa webbapplikationer. Syftet med att jämföra testverktygen var att hjälpa mjukvarutestare att välja rätt testverktyg för BDD. Att byta testverktyg är både dyrt

och tidskrävande då det ofta kräver stora ändringar i källkoden och extra träning av mjukvarutestarna. Att välja rätt testverktyg är inte lätt eftersom man måste ta många saker i beaktande, som till exempel testverktygets pris, användarvänlighet, effektivitet, egenskaper och tillgänglig dokumentation och stöd.

För att begränsa avhandlingens omfattning jämfördes endast tre testverktyg som stöder BDD, Cucumber, Robot Framework och Concordion. Eftersom betalda testverktyg ofta är dyra jämfördes endast gratis testverktyg med öppen källkod. Testverktygen valdes utifrån deras popularitet som bestämdes baserat på hur många stjärnor och förgreningar testverktygets öppna källkod hade på GitHub.com, en populär plattform för lagring av versionshistorik för mjukvaruutvecklingsprojekt. Mängden stjärnor och förgreningar som källkoden har på GitHub återspeglar ofta mjukvarans popularitet. Tidigare forskning om testverktyg och litteratur undersöktes för att avgöra vilka krav mjukvarutestare anser är viktiga när de väljer testverktyg. Testverktygen jämfördes utifrån dessa krav. De fem viktigaste kraven som identifierades var:

- Användarvänligheten
- Testrapporteringen
- Testverktygets popularitet, stöd och dokumentation
- Hur lätt testverktyget är att ta i bruk
- Effektiviteten

Kostnaden togs inte i beaktande eftersom endast gratis testverktyg jämfördes. Användarvänligheten utvärderades med hjälp av system usability scale (SUS), tidigare forskning samt antalet rader av kod som krävdes för att implementera ett testfall. SUS är en metod för att på ett enkelt sätt kunna jämföra och utvärdera användbarhetsaspekter mellan olika system. I resultaten bör det tas i beaktande att SUS-bedömningen endast utfördes av författaren av avhandlingen, vilket leder till att SUS-värdena är subjektiva. SUS-värdena ger därmed endast en bild av hur användarvänliga testverktygen är jämfört med varandra. Testrapporteringen utvärderades baserat på hur detaljerade rapporterna är och vilka format som stöds. Testverktygens popularitet, stöd och dokumentation utvärderades baserat på hur många som söker efter testverktyget på sökmotorn Google samt hur mycket

stödmaterial och dokumentation som finns tillgängligt på testverktygets hemsida. Hur lätt det var att ta i bruk testverktygen utvärderades baserat på hur länge det tog att installera testverktyget och implementera samt köra ett exempeltestfall ur dokumentationen på testverktygens hemsida. Ifall det tog mindre än 60 minuter att göra detta, ansågs testverktyget vara lätt att ta i bruk. Effektiviteten utvärderades baserat på hur länge det tog att köra testfallet. För att jämföra testverktygen implementeras ett testfall för Evernotes webbapplikation. Evernote är en applikationsprogramvara för insamling av ljud-, bild- och textanteckningar, med mera. Samma testfall implementerades och kördes med Cucumber, Robot Framework och Concordion.

Vid jämförelsen av testverktygen hade Robot Framework det högsta SUS-värdet, 85, och det krävdes minst rader av kod, 67, för att implementera testfallet med Robot Framework. Detta var tack vare Robot Frameworks kortfattade nyckelords-syntax. Av denna orsak lämpar sig Robot Framework även för mjukvarutestare som saknar kodningserfarenhet. Cucumber hade det näst-högsta SUS-värdet, 75, och det krävdes 100 rader av kod för att implementera testfallet. Concordion hade det lägsta SUS-värdet, 42.5 och det krävdes flest rader av kod, 167, för att implementera testfallet. Robot Framework och Concordion hade inbyggd, detaljerad testrapportering i HTML-format, medan Behave, Python-versionen av Cucumber som användes för denna avhandling inte hade inbyggd testrapportering. Både Cucumber och Robot Framework var populära eftersom det gjordes ungefär 6500 sökningar på Cucumber-testverktyget och 4500 sökningar på Robot Framework per månad med Google-sökmotorn. Concordion var mindre populärt eftersom det endast gjordes ungefär 20 sökningar på Concordion per månad med Google-sökmotorn. Cucumber hade mest stödmaterial och dokumentation tillgängligt på testverktygets hemsida och det var även det enda testverktyget som erbjöd interaktiva självstudieprogram på hemsidan. Både Cucumber och Robot Framework var lätta att ta i bruk eftersom det tog mindre än 60 minuter att implementera ett exempeltestfall från dokumentationen på testverktygens hemsidor. Concordion hade minst dokumentation och stödmaterial på hemsidan, vilket gjorde att det tog längre att implementera testfallet. Att man var tvungen att installera Cubano, ett mjukvarubibliotek, för att köra acceptanstester med Selenium med Cubano gjorde också att det tog en längre tid att implementera testfallet. Alla tre testverktyg var

ungefär lika effektiva. Det tog 24 sekunder att köra testafallet med Cucumber, medan det tog 27 sekunder att köra testfallet med de övriga testverktygen. För att se en större skillnad i effektiviteten krävs att ett längre testfall körs.

Resultaten av jämförelserna indikerar att både Robot Framework och Cucumber är bra val för BDD i webbapplikationsutvecklingsprojekt. Concordion var sämst i alla jämförelser förutom testrapportering och effektivitet. Robot Framework och Cucumber var båda lättanvända, det fanns mycket dokumentation och stödmaterial tillgängligt på testverktygens hemsidor, de var lätta att ta i bruk och de hade god testrapportering, antingen inbyggt eller med mjukvarubibliotek. Concordion var mindre populärt, mera svåranvänt och det fanns mindre dokumentation och stödmaterial tillgängligt på testverktygets hemsida. Concordion lämpar sig därför bäst för mjukvarutestare som har längre kodningserfarenhet och som är bekväma med att utföra felsökning.

Bibliography

[1] J. F. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, 2014.

[2] The Standish Group, "CHAOS Report", Standish Group, Boston, MA, USA, 1995. Accessed on: May., 12, 2021. [Online]. Available: https://www.projectsmart.co.uk/white-papers/chaos-report.pdf

[3] A. Axelrod, *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. Apress, 2018.

[4] J. C. Goodpasture, *Project Management the Agile Way, Second Edition: Making it Work in the Enterprise*. J. Ross Publishing, 2015.

[5] "SDLC - Waterfall Model." https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm (accessed May 04, 2021).

[6] K. Rungta, "What is Waterfall Model in SDLC? Advantages & Disadvantages," Jan. 01, 2020. https://www.guru99.com/what-is-sdlc-or-waterfall-model.html (accessed May 04, 2021).

[7] "SDLC - V-Model." https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm (accessed Feb. 17, 2021).

[8] B. Hambling, *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*. 2015.

[9] "SDLC - Agile Model." https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm (accessed Feb. 17, 2021).

[10] "What is Agile Software Development?," Jun. 29, 2015. https://www.agilealliance.org/agile101/ (accessed May 04, 2021).

[11] "Agile Manifesto for Software Development," Jun. 29, 2015. https://www.agilealliance.org/agile101/the-agile-manifesto/ (accessed May 04, 2021).

[12] "Scrum," Apr. 07, 2017. https://www.agilealliance.org/glossary/scrum/ (accessed May 05, 2021).

[13] "Extreme Programming (XP)," Jun. 14, 2017. https://www.agilealliance.org/glossary/xp/ (accessed May 05, 2021).

[14] L. Koskela, *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Manning Publications, 2008.

[15] L. Shklar and R. Rosen, *Web Application Architecture: Principles, Protocols and Practices*. John Wiley & Sons, 2004.

[16] P. McFedries, *Web Coding & Development All-in-One For Dummies*. John Wiley & Sons, 2018.

[17] T. Contributor, "What is Web Application (Web Apps) and its Benefits," Aug. 26, 2019. https://searchsoftwarequality.techtarget.com/definition/Web-application-Web-app (accessed May 05, 2021).

[18] "Website." https://www.cloudflare.com/learning/serverless/glossary/client-side-vs-server-side/ (accessed May 05, 2021).

[19] "HTML." https://html.spec.whatwg.org/ (accessed May 25, 2021).

[20] "CSS Snapshot 2020." https://www.w3.org/TR/css-2020/ (accessed May 25, 2021).

[21] "JavaScript Popup Boxes." https://www.w3schools.com/js/js_popup.asp (accessed May

27, 2021).
[22] K. Rungta, “What is Software Testing? Definition, Basics & Types,” Jan. 01, 2020. https://www.guru99.com/software-testing-introduction-importance.html (accessed May 25, 2021).
[23] “What is Software Testing? Definition, Basics & Types.” https://www.guru99.com/software-testing-introduction-importance.html (accessed Feb. 17, 2021).
[24] “Testing.” https://nuscs2113-ay1819s1.github.io/website/se-book-adapted/chapters/testing.html (accessed May 25, 2021).
[25] Testim, “What Is Test Automation? A Simple, Clear Introduction,” Aug. 06, 2019. https://www.testim.io/blog/what-is-test-automation/ (accessed May 25, 2021).
[26] “List of unit testing frameworks.” https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks (accessed May 25, 2021).
[27] J. Rasmusson, *The Way of the Web Tester: A Beginner’s Guide to Automating Tests*. Pragmatic Bookshelf, 2016.
[28] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2016.
[29] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. John Wiley & Sons, 2005.
[30] K. Rungta, “What is System Testing? Types & Definition with Example,” Jan. 01, 2020. https://www.guru99.com/system-testing.html (accessed May 05, 2021).
[31] “ISTQB Glossary.” https://glossary.istqb.org/en/search/ (accessed Feb. 17, 2021).
[32] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.
[33] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Pearson Education, 2009.
[34] H. Vocke, “The Practical Test Pyramid.” https://martinfowler.com/articles/practical-test-pyramid.html (accessed May 27, 2021).
[35] M. M. Moe, University of Computer Studies, Hpa-An, Kayin State, and Myanmar, “Comparative Study of Test-Driven Development TDD, Behavior-Driven Development BDD and Acceptance Test–Driven Development ATDD,” *International Journal of Trend in Scientific Research and Development*, vol. -3, no. -4. pp. 231–234, 2019, doi: 10.31142/ijtsrd23698.
[36] “BDD: Learn about Behavior Driven Development,” Dec. 17, 2015. https://www.agilealliance.org/glossary/bdd/ (accessed Jan. 18, 2021).
[37] G. Adzic, *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications, 2011.
[38] B. Dijkstra, “Can BDD be combined with ATDD?,” Oct. 23, 2020. https://specflow.org/blog/can-specflow-and-bdd-be-combined-with-atdd/ (accessed May 26, 2021).
[39] “59 Best Automation Testing Tools: The Ultimate List Guide.” https://testguild.com/automation-testing-tools/ (accessed Mar. 08, 2021).
[40] “Behavior Driven Development - Tools.” https://www.tutorialspoint.com/behavior_driven_development/behavior_driven_development_tools.htm (accessed May 26, 2021).
[41] “Concordion.”

https://concordion.org/questions/java/markdown/#who-developed-concordion (accessed May 05, 2021).
[42] Eficode, “Robot Framework: Past, Present and Future.” https://www.eficode.com/blog/en/blog/robot-framework (accessed May 05, 2021).
[43] J. Stenberg, “BDD Tool Cucumber is 10 Years Old: Q&A with its Founder Aslak Hellesøy,” *InfoQ*, Apr. 30, 2018. https://www.infoq.com/news/2018/04/cucumber-bdd-ten-years/ (accessed May 05, 2021).
[44] “3 open source behavior-driven development tools.” https://opensource.com/article/19/2/behavior-driven-development-tools (accessed May 05, 2021).
[45] cucumber, “cucumber/cucumber.” https://github.com/cucumber/cucumber (accessed Jan. 26, 2021).
[46] robotframework, “robotframework/robotframework.” https://github.com/robotframework/robotframework (accessed Jan. 26, 2021).
[47] concordion, “concordion/concordion.” https://github.com/concordion/concordion (accessed Jan. 19, 2021).
[48] “WebDriver.” https://www.selenium.dev/documentation/en/webdriver/ (accessed May 26, 2021).
[49] “Behavior Driven Development - Tools.” https://www.tutorialspoint.com/behavior_driven_development/behavior_driven_development_tools.htm (accessed Jan. 26, 2021).
[50] M. Gärtner, *ATDD by Example*. Addison-Wesley Professional, 2012.
[51] H. Sheth, “Selenium Python Tutorial: Getting Started With BDD In Behave,” Jul. 15, 2020. https://dzone.com/articles/selenium-python-tutorial-getting-started-with-bdd (accessed May 27, 2021).
[52] “Robot Framework.” https://robotframework.org/#documentation (accessed Jan. 25, 2021).
[53] robotframework, “robotframework/SeleniumLibrary.” https://github.com/robotframework/SeleniumLibrary (accessed May 07, 2021).
[54] “How to write and use your own custom Robot Framework Python libraries.” https://robocorp.com/docs/development-guide/robot-framework/how-to-use-custom-python-libraries-in-your-robots (accessed May 07, 2021).
[55] “Concordion.” https://concordion.org/documenting/java/markdown/ (accessed Feb. 09, 2021).
[56] “Concordion.” https://concordion.org/instrumenting/java/markdown/ (accessed Feb. 09, 2021).
[57] M. Tiitinen, "Key Factors for Selecting Software Testing Tools," M.S. thesis, Bus. Inform., MUAS., Helsinki, 2013. Accessed on: Mar. 8, 2021. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/70826/Tiitinen_Minna.pdf
[58] “How do People Select Test Automation Tools?,” Jun. 20, 2018. https://www.katalon.com/resources-center/blog/select-test-automation-tools-criteria/ (accessed Mar. 08, 2021).
[59] S. Sharma, S. Patnaik, and D. Naresh, “10 Points to Help You Choose the Right Test Automation Tool,” Apr. 02, 2020. https://testsigma.com/blog/10-points-to-help-you-choose-the-right-test-automation-tool/ (accessed Mar. 08, 2021).
[60] J. Simpson and J. Wisnowski , “Automated Software Testing Implementation Guide,”

Airforce Institute of Technology. (AFIT), Dayton., OH, USA, Apr., 2017, Accessed on: Mar., 8, 2021. [Online]. Available: https://www.afit.edu/stat/statcoe_files/Automated_Software_Testing_Implementation_Guide.pdf

[61] P. Sabev and K. Grigorova, "A Comparative Study of GUI Automated Tools for Software Testing," in Proc. of The Third International Conference on Advances and Trends in Software Engineering, Apr. 23-27, 2017, Venice, Italy [Online]. Available: https://www.thinkmind.org/articles/softeng_2017_1_20_64068.pdf. [Accessed: 23 Mar. 2021].

[62] N. Thomas, "How To Use The System Usability Scale (SUS) To Evaluate The Usability Of Your Website," Jul. 13, 2015. https://usabilitygeek.com/how-to-use-the-system-usability-scale-sus-to-evaluate-the-usability-of-your-website/ (accessed Mar. 24, 2021).

[63] S. Mcleod, *Likert Scale Definition, Examples and Analysis*. 2020.

[64] "SEO Software for 360° SEO Analysis of your Website." https://seranking.com/ (accessed May 26, 2021).

[65] "Competitors Analysis & Research Tool." https://seranking.com/competitor-traffic-research.html (accessed May 26, 2021).

[66] "Keyword Suggestion Tool." https://seranking.com/keyword-suggestion-tool.html (accessed May 26, 2021).

[67] "Best Note Taking App - Organize Your Notes with Evernote." https://evernote.com (accessed May 26, 2021).

[68] "Allure Framework." https://docs.qameta.io/allure/ (accessed May 26, 2021).

[69] "Formatters and Reporters — behave 1.2.6 documentation." https://behave.readthedocs.io/en/stable/formatters.html (accessed May 26, 2021).

[70] "Concordion Integration With Jenkins," May 09, 2014. https://shinesgio.wpcomstaging.com/2014/05/09/concordion-integration-with-jenkins/ (accessed Mar. 24, 2021).

[71] "How to generate reports in Behave-Python?" https://stackoverflow.com/questions/40763066/how-to-generate-reports-in-behave-python (accessed Mar. 22, 2021).

[72] "Robot Framework User Guide." https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#post-processing-outputs (accessed May 11, 2021).

[73] "Cucumber School - Free Online BDD Training." https://cucumber.io/school/ (accessed May 12, 2021).

[74] "Resources & Upcoming Events." https://cucumber.io/resources/products/ (accessed May 12, 2021).

[75] "Cucumber Open - Get Started with BDD Today." https://cucumber.io/tools/cucumber-open/support/ (accessed May 12, 2021).

[76] "Cucumber Open." https://community.smartbear.com/t5/Cucumber-Open/bd-p/CucumberOS (accessed May 12, 2021).

[77] "Robot Framework." https://robotframework.org/ (accessed May 12, 2021).

[78] "RoboCon 2021." https://robocon.io/ (accessed May 12, 2021).

[79] "Robot Framework." https://forum.robotframework.org/ (accessed May 12, 2021).

[80] "concordion." https://groups.google.com/g/concordion (accessed May 12, 2021).