Självständigt arbete på grundnivå

Independent degree - first cycle

Datateknik
Computer Engineering

Extending Automated Testing To High-level Software Requirements
A study on the feasibility of automated acceptance-testing

Poonam Rai
Abstract

Automated acceptance testing is the testing of software done in higher level to test whether the system abides by the requirements desired by the business clients by the use of piece of script other than the software itself. This project is a study of the feasibility of acceptance tests written in Behavior Driven Development principle. The project includes an implementation part where automated acceptance testing is written for Touch-point web application developed by Dewire (a software consultant company) for Telia (a telecom company) from the requirements received from the customer (Telia). The automated acceptance testing is in Cucumber-Selenium framework which enforces Behavior Driven Development principles. The purpose of the implementation is to verify the practicability of this style of acceptance testing. From the completion of implementation, it was concluded that all the requirements from customer in real world can be converted into executable specifications and the process was not at all time-consuming or difficult for a low-experienced programmer like the author itself. The project also includes survey to measure the learnability and understandability of Gherkin- the language that Cucumber understands. The survey consist of some Gherkin examples followed with questions that include making changes to the Gherkin examples. Survey had 3 parts: first being easy, second medium and third most difficult. Survey also had a linear scale from 1 to 5 to rate the difficulty level for each part of the survey. 1 stood for very easy and 5 for very difficult. Time when the participants began the survey was also taken in order to calculate the total time taken by the participants to learn and answer the questions. Survey was taken by 18 of the employers of Dewire who had primary working role as one of the programmer, tester and project manager. In the result, tester and project manager were grouped as non-programmer. The survey concluded that it is very easy and quick to learn Gherkin. While the participants rated Gherkin as very easy.

Keywords: Acceptance testing, BDD, Behavior Driven Development, Gherkin, Cucumber, Cucumber-Selenium
Acknowledgements

I would like to express deep gratitude to my supervisors Johan Deckmar, Dewire Telecommunication Systems and Jimmy Åhlander, Mid Sweden University for their guidance throughout the journey.
# Table of Contents

Abstract .................................................................................................................... iii

Acknowledgements .................................................................................................. iv

Terminology ............................................................................................................... vii

1 Introduction ........................................................................................................... 1
   1.1 Background and problem motivation ............................................................ 1
   1.2 Overall aim ................................................................................................... 3
   1.3 Scope ............................................................................................................ 3
   1.4 Concrete and verifiable goals ...................................................................... 4
   1.5 Outline ........................................................................................................ 5
   1.6 Contributions ............................................................................................. 6

2 Software Testing .................................................................................................... 7
   2.1 Automated Testing and Manual Testing .................................................... 7
   2.2 Testing levels .............................................................................................. 9
       2.2.1 Unit Testing ...................................................................................... 9
       2.2.2 Integration Testing .......................................................................... 9
       2.2.3 System Testing ................................................................................ 9
       2.2.4 Acceptance Testing ......................................................................... 9
   2.3 Testing model ............................................................................................. 10
       2.3.1 Software Development Life Cycle (SDLC) ...................................... 10
       2.3.2 Traditional Model .......................................................................... 10
       2.3.3 Agile Model .................................................................................... 12

3 Behavior-Driven Development (BDD) ............................................................... 14
   3.1 Evolution of BDD ....................................................................................... 14
   3.2 Features of BDD ....................................................................................... 16
   3.3 BDD Support Tool ..................................................................................... 18
       3.3.1 Cucumber ....................................................................................... 18
       3.3.2 Gherkin .......................................................................................... 20
       3.3.3 Step Definition ............................................................................... 25
       3.3.4 Selenium ......................................................................................... 27
       3.3.5 Maven ............................................................................................. 27
       3.3.6 IntelliJ IDEA .................................................................................. 27
       3.3.7 Xpath ............................................................................................... 27

4 Methodology ......................................................................................................... 30
   4.1 Proof-of-concept for automated acceptance testing .................................... 30
   4.2 Survey on the Learnability of Gherkin ....................................................... 32

5 Implementation ................................................................................................... 37
   5.1 Acceptance Testing .................................................................................... 37
   5.2 Implementation Coverage .......................................................................... 43
6 Results........................................................................................................................................44
7 Conclusions..................................................................................................................................49
  7.1 Measurement of practicability of acceptance testing written following BDD principles............49
  7.2 Measurement of learnability & understandability of Gherkin..............................................50
  7.3 Future Work.........................................................................................................................51
  7.4 Ethics.....................................................................................................................................52

References.........................................................................................................................................54

Appendix A: Survey Questionnaire for Learnability of Gherkin..............................................57
Terminology

ANSI  American National Standards Institute.
ATDD  Acceptance-Test Driven Development.
BDD   Behavior Driven Development.
CD    Continuous Delivery.
CI    Continuous Integration.
DSL   Domain Specific Language.
IEEE  Institute of Electrical and Electronics Engineers.
ISO   International Organization for Standardization.
OOP   Object-Oriented Programming.
Regex Regular Expression.
SDLC  Software Development Life Cycle.
SWEBOK Software Engineering Body of Knowledge.
TDD   Test Driven Development.
YAGNI You Are Not Gonna Need It.
1 Introduction

According to ANSI/IEEE 1059 standard [1], “Testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.” This might be a very generalized version of what software testing is. Software testing is the process of evaluating a software product or system in order to find out if it meets the business and technical requirements. Software testing is performed in various level and for different purposes. Software testing helps developers to identify problems in the software system or components of a software system and take proper measures to eliminate the problems. It can also guide the software development process, if implemented wisely. Besides assisting software developers to produce qualitative software applications, software testing also helps stakeholders to realize the business value of the software product since software testing run against a software project produces tangible outcomes that can reflect the current state of the project. It also keeps stakeholders to understand and ultimately appreciate the risk of software implementation process. And business interest is vital to a software company to sustain itself in the market.

Although there are various classification of software testing based on the testing purposes, methods, extent of testing and many more factors, software testing is broadly divided into two types on the basis of how it is implemented: Manual Testing and Automated Testing.

Automated testing is software testing performed with the help of a special software (not a part of the software being tested) [2] while manual testing is a software testing performed manually by testers. 

Automated testing is getting more and more popular as opposed to the tedious and error-prone way of manual testing. Especially in large scale software projects, manual testing may not ensure the time and cost efficiency of the software testing process and simultaneously the quality of the resulting software product.

1.1 Background and problem motivation

Automated testing is deployed on various levels of the Software Development Life Cycle (SDLC). The software development life cycle is a process adopted in order to aid towards building quality software in a cost efficient and time efficient way. It is a way of planning the process of developing a software till its ready for delivery to the user. The whole process is split into different phases. There are different ways to lead the development process referred to as SDLC model. Despite numerous SDLC models, the basic phases of SDLC can be enlisted as: software requirement analysis and definition, design, development and testing phases.
Every software project starts with the process of communicating the business requirements from business analysts with the software team. The ideas and goals, which the business analyst brings forward to a software company, need to be captured and perfectly defined. The misunderstood requirement might lead to the development of a fully functional software but otherwise with no whatsoever business value to the business clients. This situation, of course, is undesirable and can be a bad news to the software company since the amendment of the situation, in this stage, might be next to impossible depending upon the complexity of the projects. Even if it is possible, in order to fix the problem which has seeped through all the stages of software development process might need a lot of additional time and manpower. This eventually will make the software company to lose the credibility towards their clients.

Provided the significance of the correct capture of software requirements, the automation of the high-level SDLC aka software requirement analysis and definition phase is highly sought-after. An automation of this phase may ensure that features which are expected from the business side are realized.

In addition, there are high-level automation tools available that not only test the system requirements, but also assist in the process of capturing requirements right from the business analyst with the use of ubiquitous language like Gherkin. Ubiquitous language is a language that is shared by the developers, business clients, and customers. Gherkin has an English-like construct [3] which can be easily understood even by the people who have no kind of programming background. This easy-to-use requirement language provides a common platform for the business analyst along with software developer and tester to communicate, discuss, and define software features. These three very critical roles of the software development process are also known as “three amigos”. Due to the use of English-like construct to describe features of the system, the business analyst and end users are able to understand the results of the automated tests and evaluate the product for themselves.

The project will explore the feasibility of automating high-level software requirements and automate the acceptance testing of a web-application with Gherkin as a proof-of-concept. The completion of this thesis would automate the testing of software from the top level of SDLC where the requirements and specifications of a software are discussed between the business analyst, software developer and tester. Additionally, it explores the importance of using automated testing to guide the development of the software. The thesis will evaluate the learnability and understandability of automated acceptance testing by evaluating the degree of difficulty for participants with different roles as primary work role to adapt ubiquitous language (such as gherkin).

1.2 Overall aim

The introduction of BDD has reformed the software development industry. It is very much established that the importance of automated testing needs no intro-
duction to software industry. On top of it BDD has stretched the automated testing to the highest level of SDLC. Its a cherry on top! With the use of ubiquitous language such as Gherkin, BDD has given a common language for business analyst, software developer and tester to communicate the system requirements that actually have “business values”. And tools like Cucumber facilitates to automate those English-like constructs into acceptance tests. These automated tests can be an easy means for even the business analysts, stake holders and the end users to evaluate for themselves if a product really works as it should. The integration process of the highest level of SDLC into the automated software testing will be reflected in the thesis implementation. The real set of requirements provided by the company Dewire would be used. The project’s overall aim is to automate the acceptance-testing and evaluate the learnability of Gherkin.

In the thesis the possibility and degree of understanding and ability for newbie to communicate in such a structured way such as Gherkin will be explored. A small scale survey is implemented to check the learnability of Domain Specific Language (DSL) - Gherkin. The practicality of an executable specification will be explored and try to give an insight to those who are new to the whole idea and guide through the process of expressing system requirements into Gherkin features and automating these executable specification.

1.3 Scope

The thesis report starts with a short introduction to the world of software testing assuming that the reader has a basic awareness about the software development process and testing. The thesis describes the importance of automated testing on the production of high quality software and gradually lead the readers to the idea of driving the software development process with the automated testing of not “unit-test”, but acceptance-tests. The detailed knowledge of different features of BDD are presented in the theoretical chapters of this thesis. It will provide the base knowledge for readers to understand the implementation process of thesis by going through the details of the tools being used in the process.

The thesis is a case study of implementing acceptance-testing on the requirements provided by Dewire. As a proof-of-concept, the requirements of the project case study will be defined in Gherkin. Afterwards, Cucumber interprets it into automated tests and finally uses Selenium to drive the test cases in a browser. Cucumber is chosen in the project amongst many automated acceptance testing tools because it can assist the whole Behavior Driven Development process and also that it is co-developed by none other than the creator of Behavior Driven Development. Whereas Selenium is chosen as the web testing tool since it is free and Java can be used as scripting language which means that the author does not need to learn yet another programming language to automate the browser. Additionally there are easily available learning materials for Selenium all over the Internet.

The thesis is limited in scope to the automation of high level specifications from the software requirement analysis and definition phase. It will not cover the implementation of low level executable specification. The proof-of-concept only covers the development of software using BDD. The functional code for the tests will not
be written by the author, instead Dewire will pitch in to the case study keeping in mind otherwise, the resulting project will be too big for the Bachelor level Thesis. Only web applications would be taken into consideration and other platforms, such as mobile applications (like MonkeyTalk), are excluded from the proof of concept. Since the purpose of this part of the implementation is solely to provide with proof-of concept as to how to automate the acceptance testing based on the existing clients requirements. And not to show if the acceptance testing can be applied on various other type of applications (like mobile applications). The thesis also includes a survey which will give some insight as to how difficult or easy would it be for a newbie to learn Gherkin. The survey will be distributed to the employees of Sundsvall branch of Dewire which will include around 20 number of people. Quantitative research method is used to explore the learnability and understandability of Gherkin.

### 1.4 Concrete and verifiable goals

The thesis will be exploring the various aspects of Behavior Driven Development and lead the readers through the evolution of Test-Driven Development to Behavior Driven Development. The two theoretical chapters will be addressing the following questions:-

- Why Automated testing?
- What are the different ways of implementing Testing in different stages of Software Development process and how does it affect the software quality?
- What is Acceptance testing?
- Why Behavior-Driven Development?

The project is divided into two parts: implementation of automated acceptance testing in actual real world parameters such as raw requirements from the customer Telia against the implementation provided by Dewire and survey. The requirements are referred as “raw” since it is a very simple description of what the clients want from the web application. The description does not follow any syntax and in order to make them readily usable it needs to be “processed”. Both parts of the project has its individual goal which can be illustrated as below.

- **Measurement of practicability of acceptance testing written following BDD principles**

The testing implementation will be exploring the practicability of automated acceptance testing written by following BDD principle. The practicability of this implementation will be measured by the answers extracted from the following questions:-
Can existing requirements be fully translated into executable Gherkin features?

How easy is it for a low-experienced programmer to learn the acceptance testing that follows BDD principles?

**Measurement of learnability & understandability of Gherkin**

The survey questionnaires will be given to the participants for the sole purpose to measure the learnability and understandability of Gherkin. The measurement will be summarized by the answers to the following questions:-

- With how much accuracy did the participants with programmer/non-programmer as Primary Working Role answer the questions?
- Can participants with programmer/non-programmer as Primary Working Role learn and use Gherkin within 10 minutes?
- What percentage of participants with programmer/non-programmer as Primary Working Role thought that Gherkin was easy?
- What was the average difficulty level perceived among participants with programmer/non-programmer?

1.5 **Outline**

Chapter 2 and 3 have the detailed descriptions of all the knowledge that is required for the reader to be able to understand the techniques used in the thesis implementation as well as the importance of Domain Specific Language (DSL) such as Gherkin and automated testing. Chapter 4 describes the methodology used in the thesis. The reader will be thoroughly explained the roles of Gherkin, Cucumber and Selenium in the “methodology” section of the report. Chapter 5 displays how the concept described in the chapter 4 was implemented. Chapter 6 will contain the results of the project. Chapter 7 will have the conclusions drawn from the results describes in the chapter 6.

1.6 **Contributions**

The working project was provided by Dewire. The acceptance testing was done by the author. The survey questionnaire was also prepared by the author.
2 Software Testing

In general, software testing is the process of finding out whether a system or its components fulfill what it was supposed to. A software component or system component is executed for the purpose of evaluation. There can be many purposes to software testing like to detect if there are software bugs (error in programming), if it is installable and runnable in its intended operating environments, if the software component functions well when integrated to the software, if the expected output is received for the respective input, if the performance time is acceptable, if the original intent of the stakeholders were achieved, if the product is user-friendly etc. The different intents of software testing is used to define the extent of test in a system. Also known as testing levels. We'll go in details about testing levels in section 2.2. Besides providing the technical assistance to the technical parties involved in the software development like software developer, it also provides assurance of quality of the product to the stakeholders [4]. Software testing can help stakeholders to get objective and independent insight to the risk of software implementation.

Software testing is widely classified as automated testing and manual testing in terms of how the testing is performed. Automated and Manual testing will be explained in the following section 2.1. There are also various other classification of software testing based on the testing methods, testing levels (as said above, it will be explained in 2.2) and involvement of testing in SDLC (this will be explained in 2.3).

2.1 Automated Testing and Manual Testing

Automated testing is the software testing process where an automated tools execute repeatedly the pre-scripted tests on a software application or software components with the intent of detecting any breaches in the predicted outcomes [2]. It is critical to implement continuous integration (CI) and continuous delivery (CD) which in return is very critical to survive on today's highly competitive market. Continuous integration is a development practice of multiple developer working parallely on a same project but on different parts and the contribution is continuously integrated into a shared repository, may be several times a day. It helps to detect errors quickly and correction of bugs due to small change sets. Continuous delivery is a software engineering approach in which software is made available for release in a short cycle ensuring that the software can be reliably released at any time. An advantage of the approach is to obtain early user feedback. This lets the developer work as per the customer demands. It reduces the cost and time of changes to the application in production.

Whereas, manual testing is the software testing performed manually by software testers acting as the end user. It is primarily performed before the release of software. Depending on the scale of project, the testing process might be carried out
either in systematic procedure or rather in exploratory way. Small scale project can have the tester test as many of the systems feature as possible along with the additional tests derived intuitively from the tests already performed. While large scale project might not benefit with this way of approach instead they rely on a very carefully planned methodology so that they can cover all of the parts of the project. A test plan document is prepared to guide through the testing process.

Lets compare automated and manual testing.

<table>
<thead>
<tr>
<th>Scale of Project</th>
<th>Automated Testing</th>
<th>Manual Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possibility of error</td>
<td>The pre-scripted testing code will run error-free.</td>
<td>Due to the monotony of the repetitive task, humans are prone to error.</td>
</tr>
<tr>
<td>Test Execution Time</td>
<td>Automated test cases needs less time to execute.</td>
<td>Manual testing needs more time to run the equivalent number of tests.</td>
</tr>
<tr>
<td>Re-usability</td>
<td>Automated tests can be reused in the future.</td>
<td>Have to restart the process every time it needs to be tested.</td>
</tr>
<tr>
<td>Practicality</td>
<td>Not everything can be converted to automated testing</td>
<td>Virtually, every tasks can be tested manually.</td>
</tr>
<tr>
<td>Limitations</td>
<td>Automated test cannot detect visual change like image color or font size.</td>
<td>Manual test can detect visual change.</td>
</tr>
<tr>
<td>Capture of human experience</td>
<td>It is more of a robotic testing devoid of any human experience.</td>
<td>More likely to capture the real user experience since tester act as end user to explore the system.</td>
</tr>
<tr>
<td>Expense</td>
<td>Automation tools might be expensive.</td>
<td>Depending upon the scale of projects, the number of testers employed might be lesser or more which would eventually make it cheaper to more expensive with respective to automated testing.</td>
</tr>
<tr>
<td>Dependency</td>
<td>The result of automation is visible by everyone. The team don’t have to depend on only one individual.</td>
<td>Only the tester itself will witness the result of testing which will centralize the dependency to a single unit.</td>
</tr>
</tbody>
</table>

Table 1: Comparison of automated and manual testing
2.2 Testing levels

As described earlier in the beginning of the chapter 2, Testing levels refers to the extent of the test into SDLC level. According to SWEBOK [5], there are mainly four different types of testing on the basis of testing level without implying a specific process model. SWEBOK is an international standard ISO/IEC TR 19759:2005 [6]. These are unit-, integration- and system testing. Other types are classified on the basis of the objective of tests [5].

2.2.1 Unit Testing

Unit testing is performed by developer itself to test a specific “unit” of code. It checks the functional correctness of the smallest piece of testable software code in an application before integrating it to modules [7]. In procedural programming, a unit is often an individual function or procedure but also could be an entire module. Procedural programming is a programming paradigm organized on a series of well-structured steps and procedures. On the other hand in object-oriented programming, a unit is often a class but also could be an individual method. OOP is a programming paradigm organized on the concept of objects. Large percentage of defects are identified during unit testing [8].

2.2.2 Integration Testing

Integration testing is also known as integration and testing (I&T). Living to its name, it is performed after the “unit-test”-ed units are grouped together. The tests are run on these groups in multiple ways in order to identify the problems that may arise when these units are combined. Areas like inter-process communication, any kind of data sharing issue are supervised via integration testing. It makes sure that these integrated parts are ready for whole system testing [9]. There are various ways to implement integration testing like top-down approach, bottom-up approach, umbrella approach, big bang approach and so on.

2.2.3 System Testing

System testing is a testing performed on a completely integrated system with a purpose to evaluate the behavior of developed software product. It is carried out by the software technicians (mostly tester and sometimes developer) to check if the requirement specification is delivered by the system produced. All the tests that integration test could not address are covered in this stage.

2.2.4 Acceptance Testing

This is a testing performed on the client side completely oblivious of the internal mechanism. It checks if the business requirements specified by the business analyst are met. It basically tests for acceptability. It is the final stage to the product delivery. It is often referred to as the test applied on highest level of SDLC (SDLC will be described more elaborately in 2.3.1). Depending upon the choice of SDLC model acceptance testing can be done either at the end of the software development process or at the very beginning or done iteratively for every small SDLC cycle.
2.3 Testing model

The importance of software testing in software development process is critical (already described in section 2). Instead of addressing “testing” as a phase, it is getting more and more common to address “testing” as a process. Bowing to the importance of testing in the software development process, testing is now no more an accessory to SDLC, instead it can define the SDLC model itself according to the involvement of testing in SDLC. The testing model is broadly divided into two categories: traditional model and agile model. Let us explore the idea in a more simple way. But the explanation is incomplete without introducing the term SDLC.

2.3.1 Software Development Life Cycle (SDLC)

Software Development Life Cycle is commonly referred by its abbreviation SDLC in software engineering. It is often also used as a synonym to software development methodology and software development process. It is a process- a series of phases adapted by a software project to enable planned, efficient and cost-effective flow of the whole process of developing a software. SDLC, basically consists of analysis and definition, design, development and testing phases. There are numerous SDLC implementations, also known as SDLC models. ISO/IEC 12207:2008 is an international standard that establishes a common framework for SDLC, with well-defined terminology, that can be referenced by the software industry [10].

2.3.2 Traditional Model

It refers to the common practice of implementing tests after the functionality is developed before it is delivered to the customer. The very traditional well-known example would be V-model SDLC.

V-model [11] is a software development process model which is considered as an extension of the traditional waterfall model SDLC. Waterfall model SDLC is one of the earliest SDLC model to be used in software development processes in which each phases is strictly followed only after the completion of the phase above. The whole process is analogous to a waterfall, where the stages Requirements, Design, Implementation, Verification and Maintenance follow each other sequentially in a strict order. It is also known as linear-sequential life cycle model. While V-model is also known as verification and validation model, the downward linear side of V model is equivalent to basic waterfall model, comprising of Requirements, System Requirements, Global Design and Implementation phases, but instead of continuing linearly downward the flow moves linearly upward through the validation part which comprises of Unit Test Execution, Integration Test Execution, System Test Execution and Acceptance Test Execution as shown in the Figure 1. Every phases in the verification phase has its corresponding testing phases in the validation phase of the model. This means the testing starts from low-level design i.e. unit tests. And then followed by integration testing, system testing and acceptance test in the respective order. It is more common to practice all the other tests except unit tests manually. There are numerous choices of
frameworks for almost every programming languages. Some of the very well-known frameworks are jUnit, Rspec, CppTest, minitest and so on. Acceptance testing is usually manually tested by customer.

![Vi Diagram](image)

**Figure 1:** V-model [12]

In this model, the flow of process is very rigid and the testing itself is started until late i.e. only after the implementation of the whole project which makes it more difficult (might be impossible) to correct any kind of errors that might have been committed on the upper phases. For example, the requirements delivered by the business analyst might have been innocently misunderstood by the software developers. In other words, any kind of faults on the higher level or the requirement phase will simply propagate all the way downward which will result in loss of time and money. In addition to this, request for addition of features takes too long to deliver because developers are compelled to spend more times to fix the bugs rather than working on the addition of the features. Wasted time is a result of out-of-date (or completely absent) documentations, bug fixes and manual testing. In V-model, the placement of testing so late in SDLC clearly affects the efficiency of the software development. This SDLC model might be feasible in a very small scale projects where not much is at stake.
As oppose to traditional water fall model, agile SDLC, instead of running separate testing phase at the end of the software development process, testing is operated for each iterations. Agile breaks the whole system into small user functionality called user stories. User story is a short and simple description of a feature told from the perspective of the person who desires the new functionality from the system, in everyday language. It captures the “who”, “what” and “why” of a requirement (features). For example, as a user, I want to log in using user name and password.

The whole system is build up through the iterations of small increments with a minimal planning as shown in Figure 2. Unlike V-model, it enables the “continuous delivery” of working version of software product (which does not necessarily represent the complete product). Rather than predicting the hard written system requirements like in V-model, it works on the principle that the requirements cannot be captured all at the primary stage of the project (also referred as requirement volatility) [14] and is open to any kind of changes or addition of the features. Requirement volatility is the phenomenon of requirement changing over the time.

One of the very concrete agile practice include Behavior Driven Development (BDD) and Test-Driven Development (TDD) which we’ll be discussing in more elaborative length in separate chapter 3.
3 Behavior-Driven Development (BDD)

BDD represents a revolutionary leap that has shaken awake almost the entire software industry with its very promising features. Concisely, BDD is a SDLC model where the software development process is basically “driven” by the automated acceptance test so that software product with “business values” could be delivered faster. There are many principal features that distinguishes BDD from other SDLC model which will be explained in section 3.2. Before this it is very important to walk through the evolution of BDD which will help to give some insight as to what is BDD.

3.1 Evolution of BDD

According to 2011 edition of the Standish Group's annual CHAOS Report, 42% of project were delivered late, ran over budget, or failed to deliver all of the requested features, and 21% of projects were canceled entirely. In the software industry, this is not an uncommon situation, provided that software development is a complex field. So what are the significant reasons that lead to the failure of software projects? [15]

- failing to build good quality software
- failing to deliver the “right” software

There is no shortcut to successful software development. But there are various tools that can be implemented to aid the successful software development. One of these tools is "test automation" and its placement in a SDLC. The importance of "testing" is undeniable in SDLC. Many of the researches had been made in order to come up with a better SDLC model that might aid to the success rate. We have already went through traditional V-model SDLC and explore the effects of having testing on the later phase of SDLC. Let us now look into TDD where the tests are done prior to implementation and continue observing the relation between the placement of testing in SDLC and its effect on the quality software delivery. The description of TDD and its flaws will lead to an inevitable evolution towards a possible better solution to the flaws.

TDD is a test-first-code-later practice. In Test Driven Development (TDD), testing is performed before the implementation of the code. Many of the organizations adopting practices such as TDD have been successfully delivering high quality code with non-existent bugs [16], resulting in quicker feature addition.

TDD developer begins with writing an automated failing test that defines and fulfills a feature or the desired improvement. This makes the developer focus on the requirements before writing the actual functional code. After the test has been written, the test is run which of course fails since the functional code is yet to be written. This is to verify if the test itself functions as it is supposed to. Afterward, just enough functional code is written so that it passes the test. The next step is re-
factoring. In this step, the step is cleaned up for readability and maintainability. This step helps TDD improve the quality of the code [17] and as well as make sure that the code can be conveniently moved to where it logically belongs by continuously re-testing throughout the refactoring phase. The whole process is repeated until required, starting with writing test for small feature addition.

The practice of small increments in TDD provides more revertible checkpoints which supports continuous integration as well as continuous delivery. TDD gives the programmer a sense of confidence on whatever small piece of code is added to the system since the test needs to confirm that it has “pass”-ed before it can move further into the system. This eventually leads to lesser bugs in the ultimate product and lesser rework in the future. But TDD has its limitation. The large number of unit tests can be overwhelming and very difficult to maintain as the project grows in size. Besides, the practice of using the word “test” lead by the class-name of implementation code as a name of the test for that implementation code leads the tight coupling of the test to that implementation of code [18] and this eventually leads both software developer and the business unit to start considering the tests as simply intended only for developers and not of any use to business. In other words, it looses the business value. This might lead to the business unit to think testing is simply waste of time and money, resulting no support from business management. Even the developers might loose the focus and start making the tests that resembles more to a unit test that is written after the code is written rather than the requirement test before writing the code. This actually leads to a very interesting situation where principle of YAGNI might be broken. YAGNI is a principle of extreme programming motivating the programmers just enough codes to serve the functionality [19]. Anything more is simply considered a waste. The test based on the desired feature’s specification makes the developer write code focused on that precise requirements and nothing more. Thus, constriciting the codes to the requirement but the use of word “test” might lead the developer to loose the focus.

Even if TDD is perfectly implemented, the resulting software might have all the codes error-free (or almost) but error free software does not guarantee that the right software has been delivered. Besides, it might be expensive to add a feature only to know that it did not have any business values. This evidently shows the need to adopt a way to drag testing to the upper level of SDLC i.e. testing the requirement and features as the first step in SDLC.

This is solved by the introduction of BDD. BDD was invented by Dan North while he was seeking for an easier way to teach and practice Test-Driven Development (TDD) [20]. Dan North often stumbled upon the questions like where to start, what to test and what not to test, how much to test in one go, what to call their tests and how to understand why a test fails [20]. The answer was as simple as replacing “Test” in TDD simply by “behavior”. This is what gave birth to BDD [20]. Also a wonderful example of how “Getting The Words Right” makes whole lot of difference. When the tests are gotten rid of “test” from their name, and instead naming it to reflect the desired behavior automatically solves the above
problems that is often a big headache to testers. The answer to “where to start” can be as simple as starting with the test that represents the next most important system behavior. As far as “what and what not to test”, simply write the tests which represent the desirable system behavior and nothing else. Naming test with a sentence describing the desired behavior limits the test to a single behavior as you can only describe so much behavior in a single sentence! This answers the “how much to test in one go” and also “what to call the test”. Additionally, rightly chosen expressive test name helps to understand why a test fails. This finally answers the “how to understand why a test fails”. This is just a glimpse of the power of getting the words right to name the test such that the name distinctly expresses behavior of the system the test is looking over. [20]

BDD is all about “Getting The Words Right” to coach the course of thinking [20] [21]. Pursuing this idea, it was realized that BDD can benefit with the use of ubiquitous language. BDD uses simple ubiquitous language which resembles natural English like language constructs to capture the system specification via conversation and examples. The use of this ubiquitous language (such as Gherkin) helps even the business units to understand (and even construct perhaps) the behavior of the systems that they require. Hence, BDD brings people working on different roles in a software project such as business analysts, software developer and tester (also known as three amigos) to a common ground to be able to communicate with each other with as much as less communication breakdowns as possible. And let the business vocabulary permeate right into the codebase. With the help of tools like Cucumber, these requirements written in English-like language are converted into automated tests. This makes it possible that the business analysts can evaluate for themselves if the system behaves the way it “should”. In this way, continuous feedback is made possible for every feature added.

### 3.2 Features of BDD [22]

- **Acceptance-test driven development**
  BDD is an Acceptance-Test Driven Development (ATDD) practice. ATDD is a software development process based on the communication between the business, developers and testers [23]. As a first step to the software development process, tests for system requirement is written with the mutual collaboration of three amigos. The entire software development process is proceeded on the basis of these tests. These three amigos- business analyst, software developer and tester get together and formulate the “feature”-s of the system that really matter to the business. A feature is a tangible functionality of a software deliverable with which the business goals can be realized [24]. In this way, these acceptance tests ensure if the software will deliver the features acceptable by the business are achieved.

- **Collaboration between three amigos to specify the features by means of ubiquitous language**
  BDD is based on this very principle of collaboration between business
analyst, software developer and tester. There are both advantages and complication in this idea. The collaboration would help to write features that actually has business values. But then when people with different field of expertise come together, there might arise language complication. For example, software team might be an expert on some programming language but business analyst might not necessarily. So while translating the specifications delivered by the business analyst, software team might miss some very important parts of it. Or business analyst might design a solution instead of providing what they want. This will prevent from exploiting the enormous benefits that can be reaped from the cognitive diversity of the team. Also this will bind the development team to a particular solution instead of exploring other optimal ways. These hurdles have been taken off the road by BDD with the use of ubiquitous language easily understood by all of the team members. Thus, minimizing as much communication gaps as possible.

✦ Specification by example
Working together to specify features using English-like constructs is of course a step towards capturing rightly the features that has business values but it might not be sufficient. Sometimes, the features can be too ambiguous to really fully understand. This was addressed by Gojko Adzic in his book titled Specification by example [25]. To minimize the ambiguity of the specifications, BDD uses concrete examples and conversation with the business analyst to discover and define requirements. Wherever possible, these examples are automated in the form of executable specifications.

✦ Accept uncertainty
BDD is all about accepting uncertainty. Abiding with the agile methodology, BDD proceeds with the mantra that the requirements of a system or rather, their understanding of the requirements will evolve and change throughout the life of a project. No matter how carefully, specifications are written, its never hardwired. Instead of getting lost in the maze of requirement volatility, BDD focuses on continuous delivery to get early feedback from the customers and keep track whether the result was according to what their customers had wanted or not. This minimizes the impact of misunderstanding the system requirements.

✦ Executable Specifications
BDD automates the system specifications in every possible manner as possible. This is why the feature that are written to reflect the system behavior are also known as executable specifications.

✦ “outside-in” methodology
"Behaviour-driven development is an “outside-in” methodology .” [26]
As explained above, BDD starts at the outside by detecting and defining acceptable business outcomes and then, marches inwards into taking steps that would ultimately lead to the fulfillments of those outcomes. BDD automates these acceptance tests which is more correctly referred to as executable specifications.

BDD doesn't limit at the acceptance tests. BDD can also be used to guide programmer to write higher quality code. Similar to TDD, BDD can be used to write the tests to restrict the implementation code except that these BDD tests for code are better documented and more focused on technical specifications of how the application should behave. These are also known as “low-level specifications” [27].

**Living documentation**

Since the executable specifications uses easily understandable vocabulary, the reports produced after running it functions as product documentation not only for technical team but for all (eg, business analyst). This documentation gets updated every time the specifications are executed which means it will reflect the latest version of the application [28]. In other words, the documentation is “living”, updating automatically with little or no manual aid. Proper documentation helps software developer with the future maintenance tasks. Testers and business analysts can easily evaluate if the desirable features have been implemented. Stake holders and project manager can use the summary views to track the progress of the project development.

### 3.3 BDD Support Tool

BDD is assisted with the specialized support tools to develop a behavior driven project. There are many support tools available like Cucumber, Jasmine, Concor- dion, Squish GUI Tester, Jbehave, Rbehave, Rspec, Behat and so on. This project will be focused only on Cucumber framework.

#### 3.3.1 Cucumber

Cucumber is a software tool that automates the acceptance tests. Originally, it was written in Ruby programming language [22]. Due to its popularity, it has been translated to various other platforms beyond Ruby. Cucumber is more than just an automation tool for acceptance tests, it is a tool that collaborates customer with the IT team. Cucumber facilitates the acceptable system behavior aka specifications to be written in ubiquitous language which can be understood by everyone involved in the project. The ubiquitous language used by Cucumber is Gherkin. We will talk about Gherkin in more details in the proceeding section. By the use of Gherkin, it is easier to bring everyone in a common platform to be able to collaborate and strive together toward a common goal. Use of Gherkin encourages the participation of business team with the software team to write acceptance tests. The use of commonly understood language reduces the possible
misunderstanding of the desired system requirements thus helps acceptance test to reflect more of the business value.

![Figure 3: Cucumber for Single source of truth](image)

Living documentation is one of the side benefits of the Cucumber. Every time a feature file is executed by the Cucumber, the outcome displays if the current version of application fulfills the behavior described by the feature. This represents the up-to-date progress report. The tests being in commonly understandable language it works also as a documentation of the system. Unlike other kind of documentation, the documentation is updated by itself every time a new feature is updated thus, reflecting true state of the project. Last but not the least, Cucumber enables the business specifications to become the “single source of truth” [3] for automated tests, executable specifications and as well as living documentation as shown in the Figure 3.

![Figure 4: Cucumber Test Suite](image)
Figure 4 is a pictorial illustration of how Cucumber works. Now let us browse through this working mechanism in more details. When Cucumber is run, it first, parses the feature file written in a plain language called Gherkin. It checks for the scenarios in a feature file to test. A scenario comprises of a set of business rules called steps [22]. The steps in the feature files are expressed in plain text. These are merely a documentation of what behaviors are required to be satisfied by the system. But there is a need of a bunch of codes that instructs Cucumber as how to put these gherkin steps into action. This is done in step definitions. Step definitions consists of codes, in a programming language of choice (in this case Java), that carry out the action described in the respective gherkin steps. Step definition also has Cucumber annotation such as @Given followed by a regular expressions that defines a pattern which links the step definition to its respective Gherkin steps [22]. Regular expression is a special sequence of characters that describes a search pattern [31]. More about the regular expression will be discussed in the upcoming section. When Cucumber parses the gherkin steps, it will try to match the steps with any step-definition with the matching regular expression pattern. In this way, Cucumber maps these steps written in Gherkin to its respective step definitions with the help of regular expression. [22]

In other words, gherkin steps are the business facing part of the acceptance testing while Step-definition is the IT facing part of the acceptance testing that has the precise detailed instruction as to how to put these tests in to action against our application [22]. Step-definition glues the gherkin features with the application being built [22]. In this way, Cucumber is a very powerful tool that promises the delivery of the features of Behavior-Driven Development.

### 3.3.2 Gherkin

According to @2016 GitHub, Inc., Cucumber executes feature files which contains behaviors written in a language called Gherkin. Gherkin is in plain-text spoken language with a minimal set of rules that can be easily learned by non-programmers and yet structured enough to concisely illustrate the business rules.

Here is a simple example which we will be using to discuss various fundamental parts of Gherkin feature file (.feature file). [32]

Example 1:

**Feature:** User logs into account
In order to be able to get access to my personalized settings
As a User
I want to log into my account

**Scenario:** Existing username and matching password
  Given Bob is in the log in page
  And Username: Bob exists
  And Password: bobpswd matches
  When Bob logs in with password: bobpswd
  Then Log in is successful

**Scenario:** Existing username and matching password
  Given Doe is in the log in page
And Username: Doe exists
And Password: doepswd matches
When Doe logs in with password: doepswd
Then Log in is successful

Scenario: Wrong Password
Given Bob is in the log in page
And Username: Bob exists
And Password: pswd does not match
When Bob logs in with password: pswd
Then Log in is not successful

Scenario: Username does not exist
Given Hailey is in the log in page
And Username: Hailey does not exist
When Hailey logs in with password: pswd
Then Log in is not successful

◆ Description
Description is the part where any kind of further information on any keywords written. It does not have any keyword [29]. It can be followed after any of the parts of the feature file excepts steps. It can extend to any number of lines until the next keyword appears. It aids the documentation.

◆ Comments
Comment is the part followed by “#” [29]. This allows to write any kind of non-executable notes similar to description except that it does not have to be on the line just after the keywords. Comment is prohibited directly after “tags”. Although comment and description might look similar provided that they are not executed but the description is regarded as a part of the structured Gherkin document and serves as the documentation meant for the stakeholders while comments are meant as something more temporary [22]. Comments are more of a notes for testers and programmers working with the features. It is used mostly to comment out some steps [22].

◆ Feature
A feature represents a business specification [29]. Feature is simply there to provide a more non-technical high-level description of a software’s behavior without revealing the implementation details. One feature file describes only one feature and is saved in a file with extension “.feature” [33]. Generally, the feature title serves as the feature file name as well, provided that the feature’s title is converted into lower case and separated by underscores [22]. For example, feature titled “User logs into account” would be “user_logs_into_account.feature”. There are no hard core rule for naming a feature title. Feature title begins with a keyword “Feature:”. And the feature title are as short and as descriptive as possible. It is advised that the feature title states concisely a single activity of a specific role [26]. In other words, the feature title should describe an actual behavior by a user of the system so that when the test passes a particular feature, it means now the particular user can do the particular task. For ex-
ample, when the feature titled “User logs into its account” passes, it means now the user can log into its account. Hence, a passed feature can be a good reflection of what business desired functionality is implemented by the system.

If one likes to provide more information on a feature, one can do it in a line following feature title until any other keyword appears, also known as “description”. Although not mandatory, feature description can be very helpful for the documentation [22]. Feature can be described by the template: [22]

In order to <benefit of the feature>
As a <role>
I want <feature>

Although <role> and <feature> in the template might be self-explanatory, it might be not so quite obvious as to why and how to specify the benefit of the feature. It might seem to be more or less overlap with the <feature> but the difference is that the <benefit of the feature> describes the business value of the feature which advocates the main reason as to “why” do we need this particular feature.

For example,

In order to be able to get access to my personalized settings
As a User
I want to log into my account

**Last but not the least, a feature should describe only one event (or activity) [26].**

**Scenario**

A scenario illustrates a business rule [29]. As we said earlier that feature is simply the description of the software's desirable behaviors without detailing how that behavior is implemented. A software can be expected to display varying behaviors for varying situations. Each of these concrete examples of how the system should behave in a particular situations are known as scenario. So every time all the scenarios of a feature added to a Cucumber test suite pass, a new functionality is added to the system. These scenarios are in fact, the acceptance criteria for the system i.e. if the system fulfills these acceptance criteria, the system is behaving correctly else not.

A feature might have many acceptance criteria that will ensure that the particular feature has been implemented by the system or not. Typically each feature has 5-20 scenarios [22] – each revolving around a single event differing only in the context and the corresponding expected outcomes. Sometimes, a software behavior might be too broad and complex to put into a single feature file. Hence if there are more than 20 scenarios, it might be a sign to regroup the related scenarios into a separate feature file describing a particular aspect of a feature in order to support the readability. Particular aspect meaning scenarios with the same event and re-occurring contexts can be grouped into one scenario. For example,
Extending Automated Testing to High-level Software Requirements
Poonam Rai

if Feature: User logs into its account – becomes too big it can be simply broken into several smaller features:
Feature: User logs into its account with non-existent username
Feature: User logs into its account with wrong password

Scenario name is followed by the keyword “Scenario:” [22]. It is advised that the scenario name should state what is different from the other scenarios in the same feature [26]. Since each scenario is about the same single event through out the feature, the varying context should be the content of the scenario name. For example, it is not necessary to have the first scenario in the example named “User logs into its account with the existing user name and matching password and log in is successful”. The action itself is clear from the feature's title. Also that it is recommended to avoid the expected result in the title [26]. Instead the only varying context is enough to let the reader know if this is the scenario that he cares about.

The scenario name itself does not have any role on passing or failing of tests but naming it rightly has many merits like [26]

◦ The report that is received after the running of a feature file will enlist all the tests that passed or failed. The concise, expressive name can help one to be able to pick up in one look which of the behaviors of the feature works and which need to be fixed. This will save a lot of time instead of going through all of the tests to find it out.
◦ Scenario name when is self-explanatory, one does not need to go through all its steps.
◦ Any kind of future amendment regarding the change of an expected behavior in an existing scenario, simple change in “Then” steps of the respective scenario is all that is required- provided that the scenario is “well” named.

◆ Steps
Scenario comprises of a list of steps [29]. The steps should be constituted such that each scenario is independent of other scenarios so that it can stand on their own and can be executed independently. The inter-dependency might result in unexpected behavior and difficult to understand scenarios.
A scenario is recommended to have usually 3-5 steps [29]. In order to preserve the expressive readability, it is recommended to not exceed the number of steps describing the scenario [29].

Although Cucumber does not distinguish between the different steps, the steps of scenario follows a pattern which helps the expressiveness of the scenario as the readily understandable documentation of tests [29]:

◦ Description of initial context
◦ Description of an event
Description of the expected outcome

The description of initial context is preceded by the “Given” keyword [29]. If there are multiple Given steps, the ones following the first Given can begin with And or But to make it more readable [29].

Event is described in a line starting with “When” keyword. It is strongly recommended to have a single When step per scenario [29]. The presence of more than one event reflects that the scenario might need to be split into multiple scenarios [29].

Expected outcome is dictated on the line beginning with “Then” keyword [29]. Similar to Given steps, But or And can be used to describe more than one outcomes.

Also it is be noted that these adverbs has no significance to Cucumber, instead it is more of a support to create the most readable scenario [29].

Background

Sometimes there can be repeating Given steps in all of the scenarios in a feature file. These can be moved to the background section. [29] For example, “Given User is in the log in page” is repeated in both of the scenarios in the above example 1. This can be moved to the background section. Background:

Given User is in the log in page

It helps the reader to focus directly on what is unique and important about a particular scenario [22]. Along with this, it becomes much easier to modify any of these steps simply by changing at one place [22]. Background section starts with the keyword “Background:” before the first scenario [22].

Scenario outline and Examples

As we already know that BDD abides by the principal of Specification by example. It becomes more obvious when we stumble across multiple scenarios with just varying inputs and outputs that these scenarios are merely different examples used to discover and define requirements with accurate precision.

Instead of copying and pasting them repetitively, we can use Scenario outline as a template with placeholders for these re-occurring variables delimited by < and > and passing the values via Examples [22]. “Scenario Outline:” and “Examples:” are the keywords for scenario outline and examples respectively.

For example, The first two scenarios in Example 1 can be rewritten:

Example 2:

**Scenario Outline:** Existing username and matching password

**Given** User is in the log in page
And Username: <userName> exists  
And Password: <pswd> matches  
When <userName> logs in with password: <pswd>  
Then Log in is successful  

<table>
<thead>
<tr>
<th>Examples:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>userName</td>
<td>pswd</td>
<td></td>
</tr>
<tr>
<td>Doe</td>
<td>doepswd</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>bobpswd</td>
<td></td>
</tr>
</tbody>
</table>

The table in Examples sections must have a header row corresponding to the placeholder variables in the scenario outline. A scenario outline is run once for each row in the Examples, obviously skipping the header row.

**Data Tables**

Data Tables are used to pass a list of values to a step definition when the data does not easily fit on a single line of steps [22]. Gherkin allows to place these data in a table underneath a step. Let’s take in consideration these steps:

Example 3:

*Given* User is in the log in page  
*And* Username: Bob  
*And* Password: bobpswd

These steps can be re-written in the form of data tables instead of spreading it into multiple lines steps.

*Given* User is in the log in page with details:  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>userName</td>
<td>Bob</td>
</tr>
<tr>
<td>pswd</td>
<td>bobpswd</td>
</tr>
</tbody>
</table>

This Data Table is passed to the Step-definition as the last argument as type DataTable which can be converted to one of List<YourType>, List<List<E>>, List<Map<K,V>> or Map<K,V> [34].  
such that E,K,V must be a scalar (String, Integer, Date, enum etc).

**Tags**

Tags are great way to group scenarios [35]. They are strings followed by “@”. Tags are inherited from parent elements [22]. For example, tag above a Feature is inherited by all the scenarios and so on.
3.3.3 Step Definition

Until now we learned that all the desired system behaviors are captured from the business team with the joint effort of both the business and IT teams “raw” in Gherkin. But keep it in mind, the Gherkin text describing the business specification cannot be executed directly by Cucumber. Step Definitions are used to translate these Gherkin steps into executable tests. Step definition can be written in any programming language like Java, Ruby etc. It is a simple Java method except that it is followed by Cucumber annotation such as @Given, @When, @Then [22]. This annotation is passed with a regular expression to match one or more steps [22]. But keep in mind that Cucumber does not distinguish between these prepositions like @Given, @When or @Then and has no whatsoever significance when matching the step-definition to steps as long as the regular expression pattern matches the steps. In fact all of these are aliases for StepDefAnnotation [22]. This means that no matter whichever of these annotations are used to create the method, a step definition will match any Gherkin steps as long as the regular expression matches the step.

Additionally, to put in other words, step-definition is analogous to a method/function in any object-oriented/procedural programming language while step is analogous to method/function invocation. Lets make this idea more clear over the following step-definition method.

```java
@Given("^([A-z]+) is in the log in page\$")
public void isInTheLogInPage(String userName) {
    //here you write code to implement the test
}
```

Cucumber registers this particular method with the pattern: "^([A-z]+) is in the log in page\$" - defined by the regular expression in @Given annotation. So when it runs the feature, Cucumber scans for the step which matches with the patterns that it recognizes. In this case, step: Given Bob is in the log in page, is matched. The mechanism of how this step matched the regular expression pattern will be explained in more detail in the later section. When Cucumber finds it, Cucumber passes all of the captures- in this case- Bob as argument to the method isInTheLogInPage() and invokes the method.

Similarly, it also matches the other first Given steps from all the Scenarios of Example 1. So every time Cucumber comes across these steps, it invokes the method isInTheLogInPage ().

**Regular Expression**

Regular expression is used to map Gherkin steps to the corresponding step definitions. Regular expression (also known as regex) is a special combination of text string to describe a search pattern. It consists of one or more character literals, operators or constructs [31]. Any regex is composed of operators. There are numerous types of operators. **Character literals**, also known as “match-self operator” literally match to the character [36]. For example
Extending Automated Testing to High-level Software Requirements
Poonam Rai

Regex: a matches the character a literally (case sensitive)

**Character class** matches any one of a set of characters [31]. For example
Regex: [äåö] matches any of the characters ö,å or ä
Regex: [A-z] matches any character ranging from A to z
Regex: \\w matches any of the word character
Regex: \d matches any decimal digit
Regex: . Matches any character

**Anchors** are metacharacters used to mark the beginning and end of the regular expression [3].
Regex: ^I have 3 apples$ matches I have 3 apples

**Group constructs** are used to match a part of input strings [31].
Regex: I have (\d) apples will match I have 3 apples and capture 3
Regex: (?:apple) matches apple but does not capture it

**Quantifiers** are the operators that determines the number of occurrence of a character literals, grouping constructs or character class allowed in the input matching the regex [31]. Here are couple of the most frequently used quantifiers.
Regex: a* matches 0 or more instances of a
Regex: a+ matches one or more instances of a
Regex: a? matches one or zero times

### 3.3.4 Selenium

One of the most popular ways to use Cucumber is to automate the development of web applications but the interesting thing is that Cucumber doesn't know how to communicate with web application but it works well with Browser Automation tools such as Selenium.

Selenium automates browsers. Selenium WebDriver is the most popular Java library for programmatically interacting with a web application. Selenium WebDriver is the successor to Selenium Remote Control [37]. It directly launches a browser instance and controls it as per the command it receives. It can also be used to run the automation suites and tests across many environments and many browsers at a time. Selenium tests can be written in various programming language like Java, C#, Ruby and Python.

### 3.3.5 Maven

The easiest way to setup selenium project is to use Maven. Maven also known as Apache Maven, is a software project management and comprehension tool used primarily for Java projects. Based on Project Object Model file (POM.xml), maven will download the java bindings (the Selenium 2.0 java client library) and all its dependencies. [38]
### 3.3.6 IntelliJ IDEA

IntelliJ IDEA is a Java Integrated Development Environment (IDE) developed by JetBrains for easier software development to computer programmers [39].

### 3.3.7 Xpath

Selenium provides various methods to locate element like on the basis of their id, name, xpath, class name, css selector and so on. Although HTML is lesser strict than XML, both XML and HTML are markup languages. Selenium uses this property of the web pages as a leverage to use this powerful language- Xpath to navigate through elements and attributes in an XML document. Xpath uses path expressions to select nodes. Following are some of the xpath expressions [40] [41].

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Any xpath expression beginning with it indicates that it is an absolute path. Meaning that it will begin with the node at the root of the XML tree. This when used in the middle of the xpath expression would mean that the node is an immediate child of the node preceding “/”. For example, /a/b will locate all the “b” nodes which are direct descendants of root node “a”</td>
</tr>
<tr>
<td>//</td>
<td>Xpath expression when begun with // selects the nodes located anywhere in the document. When used in the middle of the xpath expression like /a//b will select all the nodes “b” that are either direct or indirect descendants of root node a.</td>
</tr>
<tr>
<td>.</td>
<td>It selects the current node.</td>
</tr>
<tr>
<td>..</td>
<td>It selects the parent of the current node.</td>
</tr>
<tr>
<td>@</td>
<td>It selects the attribute.</td>
</tr>
<tr>
<td>[]</td>
<td>Predicates are embedded in square brackets. Predicates hold the condition needed to be fulfilled in order for the nodes to be selected. Let us explore through various xpath functions that are commonly used in an xpath predicates.</td>
</tr>
<tr>
<td>=</td>
<td>//a[@b='c'] selects all the nodes “a” that has attribute value of b that exactly matches to 'c'</td>
</tr>
</tbody>
</table>
| contains(arg1, arg2) | This is a very useful xpath function that determines whether the string arg1 contains the string arg2 and returns boolean true or false. It is like a lesser-strict version of “=”.
For example, //a[contains(@class, 'x')] will select also <a class='x y z'/> since it “contains” x. |
Everything in an XML document is a node. Note that everything except text are considered element node. Text are contained in text nodes and not in element nodes.

For example, `<div> This is a text </div>` The text “This is a text” is not contained in element “div” instead it is contained in the text node contained by “div”. This why there is a slightly different way to access the text which is “text()”.

So `//div/text()` This selects the text “This is a text” inside the element div.

not() It negates the boolean value that it receives.

and “and” is used if more than one condition needs to be fulfilled in order for the nodes to be selected.

or “or” is used if either of the conditions need to be fulfilled in order for the nodes to be selected.
4 Methodology

The thesis implementation section is devoted on presenting the development tools and development process involving both the proof-of-concept for automated acceptance testing and the survey on the Learnability of Gherkin. This chapter is divided into two sections.

exploring of the feasibility of automated acceptance testing framework: Cucumber in Java version.

4.1 Proof-of-concept for automated acceptance testing

■ Development Tools

◆ Literature Study

The first and foremost step to the project was the literature study about Behavior Driven Development and Behavior Driven Development Supporting Tool: Cucumber Framework. For the study of Behavior Driven Development, “BDD IN ACTION” by John Ferguson Smart was consulted. The answer to why, what and how BDD is implemented is thoroughly explained in the book. While for learning about Cucumber Framework for Java, “The Cucumber For Java Book” by Seb Rose, MattWyne and Aslak Hellesoy was consulted. This book has very useful guidelines to use Cucumber through the whole Behavior Driven software development lifecycle as well as to apply Cucumber for acceptance testing.

◆ IntelliJ Idea

IntelliJ Idea was chosen as a Java IDE since author was working with Cucumber JVM. Cucumber JVM is a Java version of Cucumber (section 3.2).

◆ Cucumber-Selenium Framework

Cucumber is the tool that automates the acceptance tests. There are plenty of choices available for Cucumber implementations that one can freely choose as per the convenience. Step-definition was written in Java programming language as per the author’s convenience. The choice of Cucumber implementation should not affect the quality or performance of the tests. Cucumber automates the Gherkin features as executable specification by linking each Gherkin steps to their corresponding step-definition with the matching patterns defined by regular expression in the Cucumber annotation prefixing each step-definitions. There are a lots of website dedicated to regular expression. Among which, author found https://msdn.microsoft.com/en-us/library/az24scfc
Extending Automated Testing to High-level Software Requirements
Poonam Rai  2016-01-20

%28v=vs.110%29.aspx and http://metahtml.sourceforge.net/documentation/regex/ very resourceful to learn regular expression quickly. Author used the online debugger for regular expression: https://regex101.com/ to debug the regular expressions.
In order to automate the testing of web-application: Touch-point, Java implementation of Selenium WebDriver was used. While using Selenium webdriver, XPath was used to quite an extent since it is one of the most powerful way to locate web elements. Author consulted http://www.w3schools.com/xsl/xpath_intro.asp and https://www.w3.org/TR/xpath/ for learning Xpath. There are plenty of online services that helps to check the validity of the Xpath expression which can prove to be very handy in order to debug the Xpath expression since IntelliJ does not have much “support” for this service. Author used http://www.freeformatter.com/xpath-tester.html.

- Maven
Since the project has numerous dependencies to various libraries like cucumber-java, cucumber-junit, junit, selenium-java and hamcrest-all, maven was used to manage these dependencies.

- Development Process

![Development Process Diagram](image_url)

Figure 5: Acceptance Testing Development Process
The process of implementing acceptance testing on the Touch-point project of Telia is summarized in the Figure 5.
4.2 Survey on the Learnability of Gherkin

■ Development Tools

✦ Google Form
Instead of making a survey form from the scratch, author used the very convenient Google Form freely available on the internet to create the survey questionnaires.

✦ Google Sheet
The resulting responses on the Google Form can be directly accessed in the form of spreadsheet similar to Microsoft Excel spreadsheet called Google Sheet. All the data from the survey responses are enlisted in the spreadsheet for the further data evaluation.

✦ LibreOffice Calc
The data from the Google Sheet were manipulated in the LibreOffice Calc. It is an open spreadsheet program. It was used to calculate average and draw charts to display the result extracted from the data. The plain numbers or table might fail to convey very big portion of information to the reader. In addition to this, it is boring and non-appealing to the reader. Instead graphs and charts can be used to make the reader focus on the information that the author wants them to. Also it condenses large amounts of information in an effective way.

■ Development Method

✦ Quantitative Research Method
The survey was based on the quantitative research method. Quantitative research method is the systematic empirical investigation of a hypothesis’s worth [42]. The research of this genre is based on a distinctive hypothesis [43] and the statistics is used to see if the outcome agrees with the hypothesis or not. In other words, it is used to quantify an assumption by generating numerical data or data that can be transformed into statistical data from a larger sample population. This kind of research method is often denoted by the terms like “positivist” and “empiricist” [43].
The survey was focused on measuring the learnability of Gherkin. The survey required to simply find a statistic number that would either back up or otherwise, the hypothesis – Gherkin is easy to learn, which is why quantitative research method was adopted.
There were 18 participants from Dewire company who took the survey. Although it might not be a number that can give a convincing conclusion, it surely is a suggestive window to the real world around the hypothesis.
Out of 18, 10 of them had “Programmer” as primary working role and 8 had “non-programmer” as primary working role. It is to be noted that the participants with non-programmer as primary working role does not necessarily mean they have no knowledge of programming. It simply means that they do not practice programming as the primary working role in their occupation. The non-programmer constituted of Tester and Project Manager.

**Survey Structure**

The survey started with a drop-box menu where one could choose their primary working role as either programmer, tester or project manager. Then a participant should enter the current time which would be used later to calculate the total time taken by the participants to answer the survey. There were three sections following this section. Each section comprised of questions. These questions were grouped into three sets and hence three sections- one section for each set. Each set reflected different level of difficulty. The first set of questions was meant to be of beginners level- a training wheels to the participants to learn Gherkin. Which was why it only had multiple choice questions. For instance, the very first question in the first section started by showing an example of feature titled “Count”.

**Feature:** Count
**Scenario:** Count apples
**Given** I have 2 apples
**When** I buy 2 apples
**Then** I have total 4 apples

As one can easily observe that this is fairly easy to understand Feature that adds the total number of apples that “I” already have with the apples “I” bought later. This example was followed by a question which demanded a change but without changing the feature’s behavior that is instead of 2 given apples, there were 3 apples.

**Feature:** Count
**Scenario:** Count apples
**Given** I have 3 apples
**When** I buy 2 apples
**Then** I have total __ apples

The answer to this question could be chosen from the multiple options: 3, 10, 8.
choice given. Other questions were similarly composed for the rest of the survey.

The second set of questions had medium difficulty level questions preceded by a hint as how to answer the questions. The questions in this section were taken from the examples used (and created by the author) in the theory part of the project (section 3.3.2). This particular example was chosen because logging in to a system or logging into mail is a daily phenomenon we all experience at least once every day. Which means it would be more easier and relevant to relate us with the scenarios under **Feature**: User logs into account.

Finally the last set of questions were based on a web application which author had written acceptance tests for the project. The participants were not right away exposed to the questions regarding Gherkin tests for web-application instead gradually led from the beginner level Gherkin examples and the Gherkin tests for general application in order to ease the participants into learning Gherkin. The survey questionnaire is attached in the Appendix A.

**Data Extraction**

The Figure 7 below show the calculation of Accuracy Rate (in percentage) & Average Time Taken (in minutes) from the survey. The answers submitted by the participants for each questions were compared (manually) with the “correct answer”. The answers conveying the same meaning but missing the grammatical structure were still accepted as right answer since the survey did not explicitly informed the Gherkin rules for the Gherkin steps to the readers. The number of right answers then, were quantified to calculate the accuracy with which the participants answered the questions. Figure 7 and 8 shows the calculation for the overall participants. In order to get the same result for specific group like programmers, instead of whole survey data only the data from the programmers were considered and so on. The submit time stamp is automatically submitted to the Google sheet of the Google Form while the start time were manually input by the participants.
Figure 7: Calculation of Accuracy Rate (%) & Average Time Taken

The survey had a linear scale ranging from 1 to 5 (1 being very easy and 5 being very difficult) to choose from to rate how difficult the participants felt the questions were. As said earlier, each set of questions demanded level of understanding of Gherkin, each set had this question which is why it seemed only fair that the perception of difficulty level of participants for individual sets were considered separately and average was taken for all the three sets as the Average of difficulty level rated by a participant in Figure 8. Furthermore, the average of difficulty level rated by a participant did not (of course) always resulted in whole numbers like 1,2,3,4 or 5. Instead decimal number was also received. These decimal number were rounded off to nearest integer.
Figure 8: Calculation of Average Difficulty Level rated by participants & Percentage of Participants that rated a specific Difficulty Level
5 Implementation

This chapter guides the reader along the step by step process of translating the concept described in the chapter Methodology in to reality. This chapter is also divided into two sections.

5.1 Acceptance Testing

This chapter guides the reader along the step by step process of how the automated acceptance testing was implemented. As mentioned in the chapter: Methodology, author was provided with a list of requirements for Touch-point project by Telia. In this chapter, one of those requirements is taken into consideration to lead the readers through the process. The requirements were enlisted in a very simple XML spreadsheet.

---

<table>
<thead>
<tr>
<th>For a specific Svarsgrupp node the following is shown:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Title Svarsgrupp</td>
</tr>
<tr>
<td>- Number (+4681001010)</td>
</tr>
<tr>
<td>- Svarsgruppens namn (link)</td>
</tr>
<tr>
<td>- Lock icon (if opened or closed)</td>
</tr>
<tr>
<td>- Users logged in</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link forward to correct Svarsgrupp page.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Name, number, locked/closed and logged in/out status changed.</th>
</tr>
</thead>
</table>

Figure 9: Screenshot showing part of the list of requirements

Figure 9 is the screenshot showing a section of the list of requirements. The part highlighted by blue is the requirement this chapter is using to demonstrate the process formulated to strive towards the desired result.

The above requirement in Figure 9 was translated into following Gherkin features.
The highlighted requirement in Figure 9 was captured in a feature file with the title “Admin expands Svarsgrupp”. And as explained in section 3.4.2, the file was named “admin_expands_svarsgrupp.feature”. The name of the feature represents the behavior the test would be looking over which means that when this feature passes Admin is able to expand the Svarsgrupp tab. The description of the feature title – the text followed by the feature title until “Background;” reflects the business value of the feature as that the feature will help the “admin” to see the Svarsgrupp settings.

In Figure 10, couple of steps are put into Background since these steps represents the setup context for the feature which will be the same for the possible future addition of scenarios. The background steps helps to reach the admin to the start webpage (in Figure 11).

All the Gherkin steps will be explained along with the screenshots wherever possible and their counter part step-definitions. Lets start with the very first Given step in background section.

CT_REMOTE_USER=Selenium&IDP=TIWSS&MO_ROLE=CUSTOMER_SUPPORT

Note: This step was changed in the Figure 10 such that the link to the website was replaced by “website”. Because the actual link is not really of much importance as long as it points to the website. But the actual link is used again to show the use of regular expression.

Step-definition:
@Given("^Admin visits (https?:\/(\//\.)+)\$")
public void adminVisits(String website)
In the above step-definition, method “adminVisits” is prefixed by Cucumber annotation @Given("^Admin visits (https?:/\/\/\/.+)$"). When Cucumber runs, it registers the method adminVisits() with the regular expression pattern "^Admin visits (https?://\/\/.+)$". Cucumber will scan steps for patterns that it recognizes. In this case, it finds that the step Given Admin visits http://smesol-aw-test.sundsvall.dewire.com:8080/web/smesol/index? CT_REMOTE_USER=Selenium&IDP=TIWSS&MO_ROLE=CUSTOMER_SUPPORT , matches the step-def method’s pattern "^Admin visits (https?://\/\/.+)$". Cucumber will pass the http link to the method adminVisits() as argument and invoke it.

The code for how step-definition is implemented depends solely on personal choice and therefore skipped for all the following examples except for this so that the reader can have a picture of how it is implemented. For any one who is interested in the implementation code, the project can be downloaded via this link: https://github.com/PnmR/Thesis-Extending-Automated-Testing-to-High-level-Software-Requirements.git

```java
@Given("^Admin visits (https?:/\/\/.+)$")
public void adminVisits(String website) throws Throwable {
    // visit the site
    WebDriver driver = new FirefoxDriver();
    driver.navigate().to(website);
}
```

This results in the action of firing a Firefox application and the link is opened. Here is a screenshot of the web page when the link is opened.

![Screenshot of the first look of the website](image)

Figure 11: Screenshot of the first look of the website
Similarly the consecutive step in this scenario represented in Figure 11 which is:

**And** Admin goes to Admin Web AB section

The following action represented by the screenshot is executed.

Figure 12: Screenshot of Admin Web AB section

Next Step:

**And** Admin clicks on Start

Figure 13: Screenshot of after clicking on the Start
Next Step:
*Given* Admin clicks on tab Växelöversikt

![Figure 14: Screenshot of when Växelöversikt tab is expanded](image)

Next Step:
*And* Admin is checking Svarsgrupp node nr 1
When Admin expands Svarsgrupp

Next step:
Then For this node, Users logged in is visible
This step verifies if the Users node is visible or not. In this particular scenario, it is clear that the condition is fulfilled. This is the final step which confirms the scenario has passed as well as the
Feature: Admin expands Svarsgrupp
Now, the Admin can expand Svarsgrupp in the system.
5.2 Implementation Coverage

Out of the long list of raw requirements from Telia, author translated 21 of them into 16 features. Thereby, resulting with 455 tests.

Figure 16: Implementation Coverage

All of the requirements from Telia that were handled were successfully translated to Gherkin executable specifications.
6 Results

This chapter presents the objective result of the survey. These results are used later in chapter Conclusion to answer to the questions that was aroused in Section 1.4.

![Accuracy Rate](image)

**Figure 17: Accuracy Rate with which participants of different Primary Work Role answered the questions**

The above Figure 17 was derived by the method described in section 4.2 under Data extraction sub-title.
Figure 18: Average Time Taken in Minutes by participants of different Primary Work Role to answer the questions

Figure 19: Histogram of Time Taken in Minutes to answer the questions by participants with Programmer as Primary Work Role
Figure 20: Histogram of Time Taken in Minutes to answer the questions by participants with Non-Programmer as Primary Work Role

Figure 21: Difficulty Level as per the participant with Primary Work Role as Programmer
Figure 22: Difficulty Level as per the participant with Primary Work Role as Non-Programmer

Figure 23: Difficulty Level as per the participants
Figure 24: Average Difficulty level perceived among participants of different Primary Working Role
7 Conclusions

The concluding statement to the thesis would be that all the steps planned for the project including research, implementation, survey and evaluation was successfully accomplished. The two different parts of the project had their own goals. In addition to this, each goal was defined by a number of thesis questions which when answered would mean that the goal was achieved. Hence, it makes more than sense to elaborate project conclusions under the thesis questions grouped under thesis goals.

7.1 Measurement of practicability of acceptance testing written following BDD principles

Can existing requirements be fully translated into executable Gherkin features?

As explained in the section 5.2, all of the requirements from Telia that were handled were successfully translated to Gherkin executable specifications. This concludes that in this project’s case the existing requirements can be fully translated into executable Gherkin features. But as reflected on section 2.1, there are of course many areas that cannot be tested by automated testing such as user experience.

How easy is it for a low-experienced programmer to learn the acceptance testing that follows BDD principles?

As a soon-to-be graduating student of Bachelor in Computer Engineering, it is safe to categorize me as a low-experienced programmer. And I personally thought it was very easy to learn and write acceptance tests.

But Gherkin tests are designed to implement the Gherkin principles. According to which the Gherkin acceptance test should also be reflecting the behavior of the system and as well as pass as living documentation. I thought these two parts were highly challenging because Cucumber has no any ways to enforce these. For example, one can write a feature titled “Account management” which is obviously a very bad choice of name since it neither reflects any of the system behavior nor it can be used as documentation. The evidence to this is that if a feature titled “Account management” fails then it gives no whatsoever help to pin point where exactly the mistake was made and how to proceed with the solution. Further, the first and foremost purpose of using ubiquitous language to write an acceptance test is to make the test easily readable and understandable. The title “Account management” might be readable but it does not give any valuable information. So when it fails, a lot of manpower and effort needs to be spent on looking harder onto the “feature” simply to locate the problem. Despite the fact that the title is so far away from a good title for a feature, but the Cucumber does not have any
mechanism to not allow this. Good or bad, it will run the test. This will put all the responsibility on the writer. Hence, I would say from the experience I got from my project's implementation, it is easy to write an acceptance test but equally challenging to make a good acceptance test that abides by BDD principles.

7.2 Measurement of learnability & understandability of Gherkin

With how much accuracy did the participants with programmer/non-programmer as Primary Working Role answer the questions?

From the Figure 17 in chapter 6, the accuracy with which the participants with programmer as primary working role answer the questions was shown 90% and participants with non-programmer as primary working role answer the questions was 98.86%. As we can see that the non-programmer had higher accuracy rate. Point to be reminded again as explained earlier in chapter Methodology that these non-programmer may not be ignorant to programming. This result could have been induced due to the fact that every of the individual tries to implement their own version of solution to a problem from their own capabilities and experiences. The programmers, who have been limited to a coding solution of almost every problem tend to be not reassured as to how these simple Gherkin text could be implemented to perform the acceptance test. This is why some of the questions were simply skipped and some of them were almost close but not quite.

With that the average of these two which gives the accuracy for the overall participants was 94% which is still very high. This simply supports to the fact that the Gherkin is very easy to understand.

Can participants with programmer/non-programmer as Primary Working Role learn and use Gherkin within 10 minutes?

As shown in the Figure 19, 40% of the participants with programmer as primary working role were able to learn and use Gherkin within 10 minutes. While Figure 20 shows that there were none of the participants with non-programmer as primary working role were able to learn and use Gherkin within 10 minutes on the other hand this same group of participants had higher accuracy rate. This result might be showing that the experience is inversely proportional to the confidence of the participants. The group with non-programmer as primary work role are more experienced which is why they know better than to be over-confident and complete the survey quicker but with lesser accuracy.

The average time taken as shown in the Figure 18 for programmer as primary work role was 13.4 minutes, 14.03 minutes for non-programmer which makes that the average time for all of the participants was 13.68 minutes. This proves that Gherkin is very quick to learn. Even in the worst case scenario here, we can observe that the longest amount of time the participants took was 30 minutes which is still very impressive provided that the average accuracy rate was 94%.
What percentage of participants with programmer/non-programmer as Primary Working Role thought that Gherkin was easy?

This measurement was taken primarily to capture and portray the feeling of easiness or difficultness from the participants’ point of view. Although the measurement of accuracy rate and time taken gives a factual basis to establish the difficulty level of Gherkin, this measurement actually collects the direct quote from the participants as to how did they themselves feel about the Gherkin.

Since the scale of difficulty level from which participants could choose was ranging from 1 to 5, 1 being very easy and 5 being very difficult, the difficulty level 1 and 2 was treated as Easy. While 3 was treated as medium and 4 and 5 was treated as difficult.

80% of programmer as primary working role thought that Gherkin was easy and 87.50% of non-programmers as primary working role thought that Gherkin was easy. Thus, it can be established that in average 83.33% of participants thought that Gherkin was easy.

What was the average difficulty level perceived among participants with programmer/non-programmer?

The average difficulty level perceived among programmer as primary working role as role is 2.03 and among non-programmer as primary working role is 1.583. The simple mathematic average of these values is 1.83 which can be used to represent the average difficulty level perceived among overall participants. This result basically helps to conclude that it is almost too easy to learn Gherkin.

7.3 Future Work

Measure how good documentation can Gherkin features be

Gherkin features are proudly presented by their inventors as living documentation. Due to lack of time and resources this measurement could not be taken. I as a student of Bachelor in computer engineering are surely not qualified to judge if a given document is a good enough documentation for a software project. But a survey in qualitative research method, could be taken where software architect could be asked how good Gherkin features are as software documentation?, does it need any manual input to complete it? If yes, how much? How much cost would be minimized by this? How much time would be minimized?

Real case study to build a complete software using BDD methodology
Although acceptance test done in BDD style facilitates the BDD driven software development process to develop a complete software but in the project it was simply used to automate the acceptance test for the Touch-point application that was already done by the developers of Dewire. Nonetheless, the next step to this project where the proof-of-concept for this style of acceptance testing was provided, would be to perform a real case study to build a complete software using BDD methodology.

**Cucumber API Extension**

Keeping in mind the fact that BDD was created in 2008 it is still very young which means there is still room for API extension. Projects dedicated to contribute on Cucumber API is surely one of the next step to this project. At the end of this project I personally simply fell in love with the BDD methodology and Cucumber. In one of my presentation, I was asked about how can we sell this idea to increase the adoption of this. It made me contemplate my experience with Cucumber to seek for answer. And I strongly feel that if Cucumber API has more choices to report results in pictorial form like charts and pie-chart, that would upgrade the executable specification to more effective documentation. A big chunk of capital is invested into documentation, this would surely eliminate if not minimize the cost on documentation. Lesser investment for quality software is sure to motivate more business clients to participate directly on writing features, which is often the biggest hurdle to BDD methodology as most of the business client are quite adamant to it.

Finally, a sneak peek to the future- its not a very new idea of having built-in function in API like Java that it provides function to calculate square root, logarithm and so on. This enables user to not care about how the solution is calculated so for a user who is very bad with Maths can still “calculate” square root or logarithm. Similarly Cucumber API can be developed to enable a user to simply write features to get the implementation of that feature provided. This would mean in the future one could build a working software simply by writing the features that they want to have.

7.4 **Ethics**

Any kind of automated testing leads to the cut down on the job as tester. This is one of the direct side-effects of automated testing.

These automated acceptance testing can advance to the height that there will be build-in implementation for a acceptance test similar to build-in function like that function that calculates the square root, absolute value and so on. This would lead to the cut down of overall job as programmer.

There might come a time when no programmer is needed at all. An implementation could be reaped simply by writing the acceptance test without ever having to
bother how the solution was implemented (as explained in section 7.3). This of course sounds far-fetched but if this happens time might come when humanity would not be exercising their brains at all. It is not a news that the downfall of many of the civilization that ever existed on the surface of the earth was “luxury”. The luxury of not having to force the mental exertion and instead having machines to do all the tasks. Who knows the humanity will be dominated by the machines as depicted in the very popular movie “Terminator”. After all future is not that far away.
References


Extending Automated Testing to High-level Software Requirements
Poonam Rai


Appendix A: Survey Questionnaire for Learnability of Gherkin

Note: The questionnaire is shown in screenshot of the google form.

Learnability of Gherkin

The purpose of this survey is to study the learnability of Gherkin. Gherkin is a plain-text spoken language that is understood by Cucumber - a framework that supports Behavior Driven Development. Gherkin helps to describe the software's behavior without detailing how that behavior is implemented.

Primary working role

1. Programmer
2. Tester
3. Project manager

Current time (when starting this test):

Time
Part I

Following is the feature that counts the total nr of apples

Fig. 1

Feature: Count

- Scenario: Count apples
  - Given I have 2 apples
  - When I buy 2 apples
  - Then I have total 4 apples

Now let’s make changes in Fig. 1

Fig. 2

Scenario: Count apples
- Given I have 3 apples
- When I buy 2 apples
- Then I have total ___ apples

Please choose a suitable answer to fill in the blank on the above scenario in Fig. 2 such that it still agrees with the feature’s behavior of Fig. 1.

- 5
- 3
- 2

For the given example:

Fig. 3

Scenario: Count apples
- Given I have 3 apples
- When I buy 2 apples
- Then I have total 6 apples

Will the Scenario in Fig. 3 pass if it is supposed to "behave" similar to Fig. 1?

- Yes
- NO

Please choose the difficulty level regarding the questions in this part.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Easy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Very Difficult</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BACK NEXT
Provided that the feature in Fig. 4 enables User to log into their account. The scenario depicts that the expected result in the system is a successful log in when the username exists in the system and password matches.

Fig. 4

**Feature:** User logs into account
In order to be able to get access to my personalized settings
As a User
I want to log into my account

**Scenario:** Existing username and matching password
Given Bob is in the log in page
And Username: Bob exists
And Password: bobpwd matches
When Bob logs in with password: bobpwd
Then Log in is successful

**Scenario:** Existing username and matching password
Given Doe is in the log in page
And Username: Doe exists
And Password: deepswd matches
When Doe logs in with password: deepswd
Then Log in is successful

Please rewrite the scenario steps in Fig. 4 for yet another user with valid username "Olle" along with matching password "ollespwd".

An illustration as to how to answer for each steps.

**Given**

Olle is in the log in page

**And**

Your answer

**And**

Your answer

**When**
Now another scenario is added when "Bob" is a valid username but the password "pswd" that is used does not match in the system. Then the system is "expected" to respond with an unsuccessful login.

Fig. 5

**Feature**: User logs into account
In order to be able to get access to my personalized settings
As a User
I want to log into my account

**Scenario**: Existing username and matching password
*Given* Bob is in the log in page
*And* Username: Bob exists
*And* Password: bobsPWD matches
*When* Bob logs in with password: bobsPWD
*Then* Log in is successful

**Scenario**: Wrong Password
*Given* Bob is in the log in page
*And* Username: Bob exists
*And* Password: pswd does not match
*When* Bob logs in with password: pswd
*Then* Log in is not successful

**Scenario**: Username does not exist

Now please write another scenario depicting the situation when username "Hailey" does not exist in the system.

An illustration as to how to answer for each steps.

*Given*
Hailey is in the log in page

*And*
Your answer

*When*
Your answer

*Then*
Your answer

Please choose the difficulty level regarding the questions in this part.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Easy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Very Difficult</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part III

Touchpoint screenshot of Company Start page

Fig. 6
Start

![Växelöversikt](image1)

Här kan du se en översikt över din växel

![Logg](image2)

Här kan du se vilka ändringar som har gjort i din växel och vem som har gjort dem.

The scenario in Fig. 7 will check if the tab Växelöversikt expands when Admin clicks on it.

Fig. 7

**Feature:** Admin clicks on tab Växelöversikt
In order to expand Växelöversikt and observe its content
As a Admin
I want to click on tab Växelöversikt

**Background:**
*Given* Admin visits Touchpoint-website
*And* Admin goes to Admin Web AB section
*And* Admin clicks on Start

**Scenario:** Tab Växelöversikt is clicked
*When* Admin clicks on tab Växelöversikt
*Then* Tab Växelöversikt expanded
*And* Admin Web AB tree structure is shown
Similarly this new scenario in Fig. 8 will check if all 3 Menystyrning are shown when the tab Växelöversikt is clicked.

Feature: Admin clicks on tab Växelöversikt
In order to expand Växelöversikt and observe its content
As a Admin
I want to click on tab Växelöversikt

Background:
Given Admin visits Touchpoint-website
And Admin goes to Admin Web AB section
And Admin clicks on Start

Scenario: Checking if all Menystyrning are shown
Given There are 3 Menystyrning
When Admin clicks on tab Växelöversikt
Then Under Admin Web AB, all Menystyrning are shown

Screenshot when tab Växelöversikt is clicked

Now please write the similar Scenario: Checking if all Svarsgrupp are shown - given that there are 3 Svarsgrupp

Given

Your answer
When

Your answer

Then

Your answer

Please choose the difficulty level regarding the questions in this part. *

1 2 3 4 5

Very Easy Very Difficult

BACK SUBMIT