2014

Semi Automated User Acceptance Testing using Natural Language Techniques

Arvind Madhavan
Iowa State University

Semi Automated User Acceptance Testing using Natural Language Techniques

by

Arvind Madhavan

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Simanta Mitra, Co-major Professor
Carl K. Chang, Co-major Professor
Jin Tian

Iowa State University
Ames, Iowa
2014

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

UAT: User Acceptance Test

NLP: Natural Language Processing

GUI: Graphical User Interface

R &PB : Record and Play Back

RTF: Robot Test Framework

SUT: System Under Test

POS : Parts of Speech

SQL: Structured Query Language

XML: Extended Markup Language

BDD : Behavioral Driven Development.

API: Application Package Interface.

CSS : Cascading Style Sheet

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr.Simanta Mitra for the continuous support of my study and research, for his patience, motivation, enthusiasm, and immense knowledge. I could not have imagined having a better advisor and mentor for my Masters.

I would also like to extend my appreciation to my co-major professor Dr. Carl K. Chang and my committee member Dr. Jin Tian for their help and support. I also thank the department faculty, and department staff who have guided us for

A very special thanks to Ramiya Venkatachalam and my brother Raghunathan Kothandaraman for the constant encouragement and support, which helped me to stay motivated .In addition I would like to thank my Iowa State friends–Arunachalam, Deepan, Gowri Shankar, Guru Prasad, Sunil Kishore and my best buddy Naveen for making this journey not only possible, but a memorable one.

Last but not the least; I would like to extend my gratitude to my parents, Mr. Madhavan Srinivasan and Mrs. Rama Madhavan for their unbounded support and love without whom this journey could have been not possible.

ABSTRACT

User Acceptance Testing is typically the final phase in a software development process in which the software is given to the intended audience or domain experts. These domain experts know the functional requirements of the application and write user acceptance tests (UAT) in their natural language. A normal UAT test case in English typically follows an imperative sentence structure, i.e. a sentence that gives advice or instructions, or that expresses a request or command.

We propose a methodology to write UAT test automation code using natural language processing techniques on test scripts written in free form English text by using the assumption that test cases are written in an imperative style. We have also built a proof of concept tool, the Autotestbot, to demonstrate the feasibility of our idea. In addition, with the help of ***Autotestbot,*** we also demonstrate the feasibility of our proposed approach to semi-automate the time consuming and cumbersome manual UAT test code generation process. The scope of this thesis is restricted to automating Web applications.

CHAPTER 1 INTRODUCTION

1.1 Background and Motivation

Software Testing is a systematic procedure for checking a program or application with the intent of finding bugs [31][51]. User Acceptance Testing (UAT) is typically the last phase of the Software Testing process where the end product is assessed for its correct business usage. It is a crucial part of the overall testing process of an application.

The first step in the UAT development life cycle is designing test cases or user scenarios for the real-world usage. Typically, business users write these in a simple language (mainly English). It is natural to ask why natural language test cases are used. The answer is that system test cases are most commonly created by non-developers i.e., business or domain experts, who may not possess the technical skills required for coding test cases in a programming language, but can represent their thoughts in natural language fashion. Natural language incurs no training overhead [49]. The objective of this research is to help these business users or domain experts to generate automated user acceptance test code from natural language test cases.

Research has shown that UAT test automation can be achieved through "Keyword" based User Interface (UI) test framework [24] where Keywords can be used as links to programs that automate test cases. The basic idea behind the Keyword based UI framework is to separate test case design and test code generation [22]. Separating the test automation makes it readable for non-technical personnel or domain experts. The Robot-Selenium framework [40] is an example of such a framework.

However, current tools, such as Cucumber [7] or Robot Test Framework [24], are constrained in that the user is forced to use the keywords pre-specified in their framework to create higher-level keywords. Our approach is to reduce this overhead by not creating excessive higher-level keywords and instead allowing free form test cases. To achieve this, natural language processing techniques have been exploited.

1.2 Problem and Approach

Automated testing is more effective than manual testing with respect to accuracy in regression tests [21]. It minimizes the margin for errors. Current Test Automation Experts manually convert written UAT test cases into functional test code using test automation tools. These tools require an extensive knowledge of the scripting language to create functional tests. Unnecessary time is expended in learning the details of the scripting language, writing the scripts, and then debugging the scripts.

The main objective of our research is to automate this time-consuming and cumbersome manual testing process by abstracting functional instructions in Natural Language and mapping them to corresponding test automation code. As a result, no executable code needs to be developed by the business user. A secondary objective of the research is to develop a comprehensive test corpus so that test scenarios across the same domain functionality can be reused.

Our goal is to generate a test automation class file from the English UAT test cases. For this we used POS tagged test cases as our knowledge base. About hundred sample test cases were collected from Internet from Quality Assurance (QA) forums and blogs and were preprocessed manually to remove ambiguous data. We then manually tagged these test cases

to build our test cases corpus. We also built a test language model specific to UAT testing and use that as reference to process untagged test cases. The user is to write test cases in a specified spreadsheet format. Our proof of concept tool analyses these test cases by appropriately tagging them and generates keyword tuples for every test case. The generated keyword tuples are eventually converted into Selenium automation class file that are run on the Firefox browser.

The scope of our project is restricted to browser based applications and cannot be extended to non-web applications .Our keyword repository is a subset of Robot Selenium Framework [24][40]. Hence any keywords or actions that do not reside in this keyword repository would fail. The scope is also restricted to the Object recognition capabilities of the Selenium Web driver [55]. This means to test automate an application with third party plugins (like Microsoft Silverlight), it requires additional code to be written. Our research covers basic Web, JavaScript, and AJAX applications. Also the browser has been restricted to Firefox at the moment for proof of concept purposes.

1.3 Requirements of the Test Framework

The objective of the thesis is to present a methodology to develop a UI test framework where natural language scripting can be used for test automation [41]. Fewster and Graham's paper on Software testing Frameworks helps us to identify the basic goals in developing a test automation framework [21][24]. To achieve these goals we first define a set of requirements as specified to be satisfied by the framework.

I. User should be able to write test cases in free form natural language (English).

II. The framework should generate the automation code for the test user

III. The generated test automation code can be executed in a browser.

IV. The code should be available to the user to easily understand. It should satisfy object-oriented principles.

V. Maintainability of the code.

VI. The code can be reused later for addition or deletion of tests. This should not affect the existing code.

The rest of the thesis is organized as follows. Chapter 2 provides a literature review of the UI Test frameworks commonly used. Natural Language based Test Automation techniques have been discussed. Chapter 3 is the crux of the thesis. It documents the System architecture for meeting the requirements specified above (for detailed requirements refer appendix A1). It also presents our approach, which includes the development of POS tagged custom Test Case Corpus, preprocessing of inputs, training, testing, and automated generation of test codes for a particular web application. Chapter 4 evaluates the feasibility of our approach. Finally, we present our conclusions and discussions on future work in Chapter 5.

CHAPTER 2 LITERATURE REVIEW

This chapter is divided into three major sections. The first section provides a brief discussion on the need for UI test frameworks. The second section reviews the different UI automation techniques with examples of frameworks that support natural language test scripts. Finally, the third section gives an introduction to POS tagging.

2.1 Need for UI test frameworks.

User acceptance Testing (UAT) helps end users to validate and verify the behavior of the final state of the product. UAT level testing happens by running end to end scenarios on the User Interface (UI) of the product .A product that is heavy GUI based has to be UI tested repeatedly to elude bugs due to regression i.e. so that changes in the software does not introduce new faults. Hence, manual testing is not efficient in repeatable UI test execution [38]. Therefore, these manual UAT tests are converted to automated tests for which some kind of a UI test framework is required.

The real need of building a UI framework is about maintainability of the UI tests. Writing a suite of such tests in an automated fashion that are maintainable is virtually impossible and expensive without using a UI test framework [57]. A UI test framework aids the tester to analyze test outcomes and report the results in an effective manner [7]. A UI framework helps the tester to design, add, delete acceptance tests, and monitor test results easily [41]. Other important benefits include adhering to a standard list of specifications (or test plan) for the product.

2.2 Review of UI Automation techniques

In this section we shall give an overview of UI automation techniques used for UAT test automation and discuss prior works in Natural language based UI automation.

2.2.1 Record and Playback

The first step towards basic automation in any GUI based test tool is record and play back option. We decided to include this in our literature review, as most non-programmers tend to use Record & Play Back tools (R &PB). R & PB tools are definitely attractive at the first instance for the reason test case execution happens on a single click of the record button. The user has to initially set the tool in a record mode that records the list of actions, which are then replayed back. R&PB tools come in handy when the user does not have programming skills to write automation scripts [29]. Some examples of Web testing tools that can enable testers to record tests and playback are Selenium IDE, Microsoft Visual Studio Coded UI [17], and Test complete [27].

The use of these record and play tools has lot of disadvantages such as lack of code reusability, maintainability, consistency in test execution. One of the biggest drawbacks in using Record and Playback is the scalability. When a test is automated using recording, script lines are generated. John Kent paper on record and play back tools [21] give us the following relation:

$$Lines\ of\ Automation\ code\ \propto Number\ of\ tests\ to\ automate.$$

The more tests you record, the more automation code you have to maintain until you reach a point where the maintainability is nearly impossible [21]. Also the recorded script can only work under exactly the same conditions as when it was recorded at the first time, which makes it less reliable.

Considering the drawbacks of Record and Play Back Tools, a more matured approach evolved which was separating the test data from the automation script. This is called the Data Driven Approach [2].

2.2.2 Data Driven and Keyword Driven UI Frameworks

In this approach the test data is held in separate files or data tables and the automation script works on the test data. For a UI based Data Driven approach the UI script is tested against a user interface that requires validation for variant data A sample scenario would be to validate login credentials for different users. When the tested system changes it is easier to change the test code as it is separated from the test data [39].

An example of Data Driven based UI framework is the Microsoft Coded UI Test [17] where data is stored in XML, Excel Worksheets, or SQL, and the framework runs UI scripts as unit tests. Other examples of Data Driven UI frameworks are Fitnesse[12], HP-Quick Test Professional and Test NG [27].

The biggest drawback of Data Driven Frameworks is that the overall functionality of the application can never be tested thoroughly [2]. Only variants of test data helped in testing specific functionalities rigorously. Also writing the test part requires knowledge of programming which makes it difficult for business users.

This gave growth to a better approach where the test input data sheet included directives for the test in form of keywords. This mean the same spreadsheet input of test data included executable test cases with the addition of keyword column. This was called the Keyword Driven approach. Keyword-driven testing has all the same benefits as data-driven testing[35]. The main objective of Keyword based approach is to allow non-programmers to write tests as well with the help of keywords. Keywords are basically English words, which performs an action. 'Click','Enter','Type' are all examples of Keywords. With the help of the Keywords and the test data, a complete UI test framework can be built that is maintainable and scalable.

Keyword Driven UI frameworks can be built using existing tools that provides us with the keyword API or library. Quick Test professional (QTP) is a VBScript based tool that supports keyword driven testing. Open2TestFramework is a framework that was built on top of HP Quick Test Professional (QTP).

Selenium consists of a suite of tools that can be used for Keyword based browser automation. The latest version of Selenium, Selenium Web driver [56] provides the user with set of API's or libraries to build Keyword based UI Frameworks. Examples of frameworks built on top of Selenium are Xebium [55], Robot-Selenium [41] and Cucumber –Selenium [7].

Watir [49] is a ruby based tool that automates Web applications. Robot Framework is a keyword-driven framework for User Acceptance tests. Watir-Robot [54] is a Keyword based framework, which uses Robot Keywords for functional web testing. With the introduction to

Keyword based UI Test frameworks the next section reviews the literature on frameworks that enables natural language test automation.

2.2.3 Natural language based Test frameworks

In this section we review some of the most commonly used UI test frameworks using which natural language tests can be used to write UAT test automation.

Cucumber [7] is an extremely powerful integration-testing framework for Acceptance Test Driven development. The idea is that you write your “user stories” in a language called Gherkin. Gherkin [59] is a business specific language that lets you describe software’s behavior in English but specific to cucumber. The user then maps it to a custom based Ruby code that executes the "user stories." The ultimate goal of Cucumber is to communicate the behavior of your system to everyone involved in a project, making it possible to write specifications in English, or other natural languages .In this way, the specs are executable, but they are also readable by non-programmers, making it easier to discuss at meetings, or in documents. However, to use cucumber we need to know its business readable, domain specific English language.

The Framework for Integrated Test (FIT) is an open source framework implementation for a table-based acceptance testing approach. FitNesse is an HTML front end to Fit [12]. FitNesse lets customers and analysts write “executable” acceptance tests using simple HTML tables. Developers write “fixtures” to link the test cases with the system under test (SUT). "Fixtures" are usually java classes [32] and is an interface between the Fit framework, test cases, and the SUT. The fixtures act as the test engine, which drives all the logic behind execution. Xebium [55] is a FitNesse framework written on top of Selenium for User

acceptance testing. Xebium combines FitNesse's powerful features with the browser testing ability of Selenium.

In [48] the author provides a proof of concept model to generate code from test scenarios written in English. Natural language techniques have been used to understand these test cases. A given test scenario is analyzed from its step definition using POS tagging methodologies. The code skeleton is then derived from the POS tags. The approach uses WorldNet dictionary to extract POS tags and the user interacts with the system to generate the code. The commonality with our approach is that we use POS tagging for information retrieval. The difference is that we do not use commonly used English corpuses. We construct our own test cases corpus to extract POS tags instead of using the WorldNet dictionary.

In the open source community, Robot Test Framework [24] is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). The biggest inspiration for our research has been the Robot Test Framework (RTF). RTF is a Python-based keyword-driven test automation framework for acceptance level testing. With RTF, the user has the ability to create test cases in an easy to use tabular syntax[7]. Users can create new keywords from existing ones using the same simple syntax that is used for creating test cases. Test results are presented in an easily understandable HTML format. This project was developed at Nokia Siemens' and is used extensively within their network. The project was made open source later and has many users outside the company. It is continuously developed and its base of keywords is growing.

The NLP approach in UAT test automation highlights the fact that users are allowed to

write test cases in their own natural language format. Robot test framework, and other keyword driven Natural language based Test frameworks force the user to build keywords on top of their inbuilt keywords. In our approach, we have a keyword-based framework as similar to the other tools except that we map these keywords intelligently rather than forcing users to use only the existing keywords.

2.3 Natural language Techniques for Acceptance Testing

English grammar categorizes a word's importance according to their role in sentences. However, many words take multiple meanings based on the context and its grammar. For instance "Like" can be a verb or a proposition. Similarly "book" can be a noun or a "verb". Part-of-speech tags give us significant amount of information about the word and its neighbor's [43] and are really useful in understanding the context.

Let us define the definition of POS tagging in mathematical terms.

Given a sequence of words $W=w_1 \ldots w_n$, *in a sentence {S} we want to find the corresponding sequence of tags* $T=t_1 \ldots t_n$, *drawn from a set of tags {T}, which satisfies the below equation .*

Equation 1:

$$S = \max_{t1\ldots tn} P(t1\ldots tn \mid w1\ldots wn) = \max_{t1\ldots tn} P(T|W)$$

We need to find T that maximizes the Equation 1.

By Bayes rule and chain rule of probability [45] POS tagging can be approximately deduced as below.

$$P(T|W) = P(W|T)\ P\ (T)\ /\ P(W)\ \approx\ P(W|T)\ P(T)$$

$$P(T)\ P(W|T)\ \approx\ P(t1)\ P(t2|t1)\ \ldots\ P(tn|tn-1)\ P(w1|t1)\ P(w2|t2)\ \ldots\ P(wn|tn)$$

$$Where\ P(t_i|t_i-1) = c\ (t_1-1, t_i)\ /c(t_i-1)$$

$$P(wi\,|\,ti\ ) = c(wi, ti)/c(ti)$$

Where , $c(t_i)$- frequency of the tag t_i in the corpus

$c(w_i, t_i)$- frequency of w_i with tag t_i in the corpus.

$c(t_{i-1}, t_{i)}$ = frequency of a tag t_{i-1} with an after tag t_i in the corpus.

Parts of speech tagging can be achieved by developing POS tagger models. The development of a reasonably good accuracy POS tagger can be categorized broadly in two ways: rule based and learning based.

Rule based Method: This consists of writing an exhaustive set of rules based on lexical and other linguistic knowledge [5]. It is very costly and time consuming to develop a rule based tagger [36].

Learning Based Method: This consists of training on human annotated corpora and using machine-learning techniques as Hidden Markov Model, Trigram Tagger, and other statistical taggers [4]. Learning based approach is considered effective [36] considering the amount of human expertise and effort involved.

We chose the Learning based method to tag our test cases. However, there were no annotated corpora readily available for developing our tagger. Hence we developed our own "test cases corpus" and trained our POS taggers on this corpus.

In the next chapter, we present our overall approach and also the design of our proof of concept tool, the Autotestbot.

CHAPTER 3 APPROACH AND DESIGN

In this chapter, we describe our overall approach and the design details of our proof-of-concept test generation tool *Autotestbot.* This tool automatically takes as input user acceptance test cases written in natural language (English) and generates as output test code using a keyword driven test framework built on the top of Selenium Web driver framework. First, we give an overview of our overall approach. Next, we describe the system architecture for *Autotestbot*. After that, we describe each of the three phases of our approach in separate sections.

3.1 Overview of our approach

Our main idea is that there are abstract linguistic rules that implicitly govern written business user acceptance tests and these linguistic rules can be derived using NLP techniques. Once these rules have been derived, new tests can be processed by application of these rules and automatically converted to executable test codes.

Our approach consists of three phases. The first two phases consist of semi-automated steps that need to be done only once for a specific domain. These are used to setup the test framework and get it ready for use. The third phase consists of several automated steps that take in input UAT tests written in natural language and generates test code that runs on Selenium Web driver.

In the first of the two setup phases, we build a tagged repository of user acceptance test cases. We do this by first collecting test case samples from similar applications and manually getting them ready for processing. Then these samples are processed using NLP

techniques and the different parts of speech (POS) are tagged for each test case sample. This tagged repository of test cases is the knowledge base for our tool and becomes the "*test cases corpus*". Details of phase one are presented in Section 3.3.

In the second setup phase, we create a dictionary of mappings from actions (i.e. verbs) in sample test cases to keywords borrowed from Selenium Web Driver framework, and also add a linkage to generated python code for the specific keyword. This is our "*keyword map*" dictionary. During the conversion phase (the third phase), this dictionary is used by the application to select the appropriate python code to call when it recognizes a verb in the user written test case. Details of phase two are presented in section 3.4. After the *test cases corpus* and the *keyword map* dictionary are created, the system is ready to process new UAT tests in an automated fashion.

In the last phase, the *conversion phase*, the *Autotestbot* Test system first takes as input the UAT tests written by business users and uses the trained "*test cases corpus*" to POS tag the input tests. Next, it applies the *keyword map* dictionary on these POS tagged inputs to convert the test cases to keywords. Finally, the system uses these keywords to create test code fragments, appropriate test setup and teardown methods, and a test driver that can be executed by users. Details of phase three are presented in section 3.5.

3.2 System Architecture

The overall architecture used in our proof-of-concept tool is similar to any other NLP based learning application where there is an NLP based language model, a Corpus (or data

set) to train the model, a test data set which is usually a portion of the training data set, and code modules that are built using existing Natural Language Tool kit libraries.

We use the NLTK tool kit available from the University of Pennsylvania for text preprocessing which includes sentence tokenization, word tokenization, tagging of words, and extraction of words as (word, tag) tuples. We chose Python for our development work because the NLTK kit is written in Python. Our automation keywords are built for the Selenium Web driver framework that automates testing in the Firefox browser environment.

The system architecture diagram for *Autotestbot* is shown in Figure 1. There are two repositories, the *test cases corpus* and the *keyword map*. Also, there are four main code blocks:

1. **Preprocessor Module :**This module reads the input spreadsheet of test cases and extracts tokens from each test case.
2. **POS tagger Module:** This module reads tokenized test cases and uses the *test cases corpus* repository to assign appropriate parts of speech tags to these tokenized test cases.
3. **Keyword Mapper Module:** This module takes tags and uses the *keyword map* dictionary to retrieve the appropriate Selenium Action method to be used during generation of test code.
4. **Code Generator Module:** This module generates the test code in Python. A python class is generated for every test suite. Every test case is a call to a Selenium web driver method with an assertion added to check actual results against an expected output.

The input to the system is a spreadsheet of test cases written in natural language and the output is a test suite (a Python based class file that runs in a Firefox browser.)

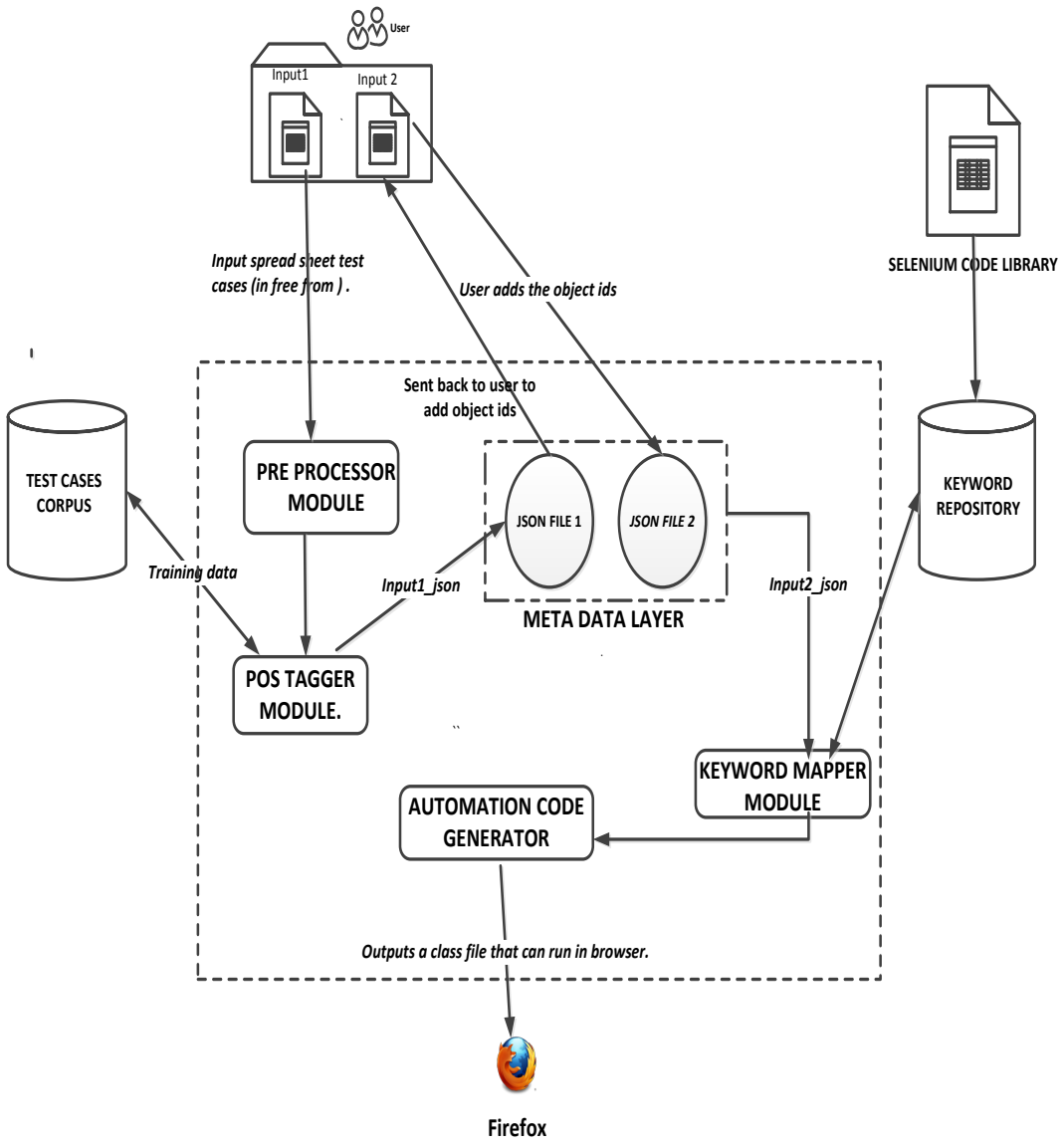

Figure 1:System Architecture diagram of Autotestbot

3.3 Phase 1 - Creation of test cases corpus

The goal of this phase is to analyze existing business user acceptance tests by using NLP techniques so as to derive abstract linguistic rules that govern these written tests and then to store these rules in our test cases corpus.

Data available on the Internet has been successfully used as training data for corpus development for many NLP applications (See [16][34]) because most of the documents are written in a machine-readable format. We too use the Internet as a resource to obtain training data. To develop a proof-of-concept prototype, we narrowed down the subject of our research to testing the login functionality of an email application. We searched for test cases (written in English) to test this functionality and retrieved about a hundred test cases from the Internet, manually corrected errors, and clustered them in an organized format. Test cases were collected from various [19] forums such as Quality Assurance forums, Blogs, Test Tutorial Sites, and Open source contributors.

Table 1:Modified Upenn Tag Set for our research

POS tag	Abbreviation	Example word
CD	cardinal number	1
CC	coordinating conjunction	and
NN	***noun, singular or mass***	***input ,button***
NNS	***noun plural***	***doors***
NNP	***proper noun, singular***	***Username, password***
RB	adverb	however, usually
VB	***verb, base form***	***Clicks***
VBG	***verb, gerund/present participle***	***pressing***

POS tag	Abbreviation	Example word
VBN	***verb, past participle***	***expired***
VBP	***verb, sing. present, non-3d, 3rd person sing. present***	***try***
VBZ	***verb***	***enters***
MD	modal	could, will
JJ	adjective	wrong
JJR	adjective, comparative	bigger
IN	preposition/subordinating conjunction	in, of, like
DT	Determiner	the

We used part of the gathered test cases to train the Natural Language Tool Kit (NLTK) and used the remaining part for testing purposes. In the following subsections, we first describe our modifications to the NLTK kit available from the University of Pennsylvania and then describe the training phase.

3.3.1 Modifications of NLTK

Two types of modifications were made to the Natural Language processing Toolkit (NLTK) obtained from the University of Pennsylvania. The first was customization of the language model used to process parts of speech of the test cases. The second was modification of some of the rules for tagging.

Language model

Any POS tagging task involves a language model (the terms and the vocabulary for the domain) as a base. The tags assigned to a sentence for a particular language depends on the language model for that language. We have defined our own custom tags for our purposes to

form our *Test Language Model.* The author has designed this language model after wading through several test cases for different scenarios from our sample data. The first stage of tagging our test cases is done by the default NLTK ‘s -Penn Set Language model. The Penn Tree bank proposes a standard set of 36 tags to identify parts of speech for English [28]. We have modified these tags to suit our purposes.

Consider the example sentence "*User enters Password and press tab key*". The POS tagged by the defaults NLTK is "User/NN enters/VBZ Password/NN and/CC press/NN tab/NP key/NN ". The POS tagged after our modification is "User/NN enters/VBZ Password/**NP** and/CC press/**VB** tab/NP key/NN ". In the example, ***Password*** is tagged as NN in general usage, but it is an object in our context, a proper noun, and hence marked as NP. Similarly, ***press*** is tagged as NN, but it is an action verb (VB) in our context (VB). The tags that have been most used in our research are shown in Table 2.

Tagging Rules

By plotting the frequency distribution of tags for the test case samples we find that NN, VBZ, NP, and VB are the most frequently used tags. There are four tagging rules corresponding to each of these tags.

Rule 1. Identification of common nouns NN. Generic objects in the system, for example, input fields, radio buttons, and text areas, will be identified as Common nouns i.e. ‘NN’.

Rule 2. Identification of Proper Nouns NP. This is used to tag a noun that refers to a unique identity, i.e., for identifying objects uniquely in the Application. For example, a "Login" button field would be identified as NP tag. However, there are certain cases where

the "Login" button should be treated as Login Action (VB) instead of NP. These rules have to be taken care of during the training phase only.

Rule 3: Identification of VB or VBZ. Most test cases are of imperative or instructional type. The action verbs in the singular or plural form are the actions that drive our Automation script on the proper nouns and our main focus is to identify only 'Action Verbs' Forms. For example, navigate to, go to, type, press, enter, click, and clicks.

Rule 4: Identification of Adjectives JJ. Adjectives play an important role in understanding the negative or the positive connotation of the test case. The role of the JJ tag is to describe the type of proper noun or the noun. For example, consider the sentences "*Enter the wrong password" and "Enter the correct password"*. The significance of JJ is evident as it alters the meaning of the test case in the two examples.

3.3.2 Training

The architectural diagram in Figure 1 helps us in understanding the steps involved in creation of a tagged test case corpus. A general representation of the POS tagging process is depicted in Figure 2. It's a three Step Process. The steps are highlighted with a dotted boundary in the figure. The inputs to the process are raw samples collected from the Internet. The output of the process is the tagged "Test Cases corpus" that consists of two files –Test Steps and Expected Output.

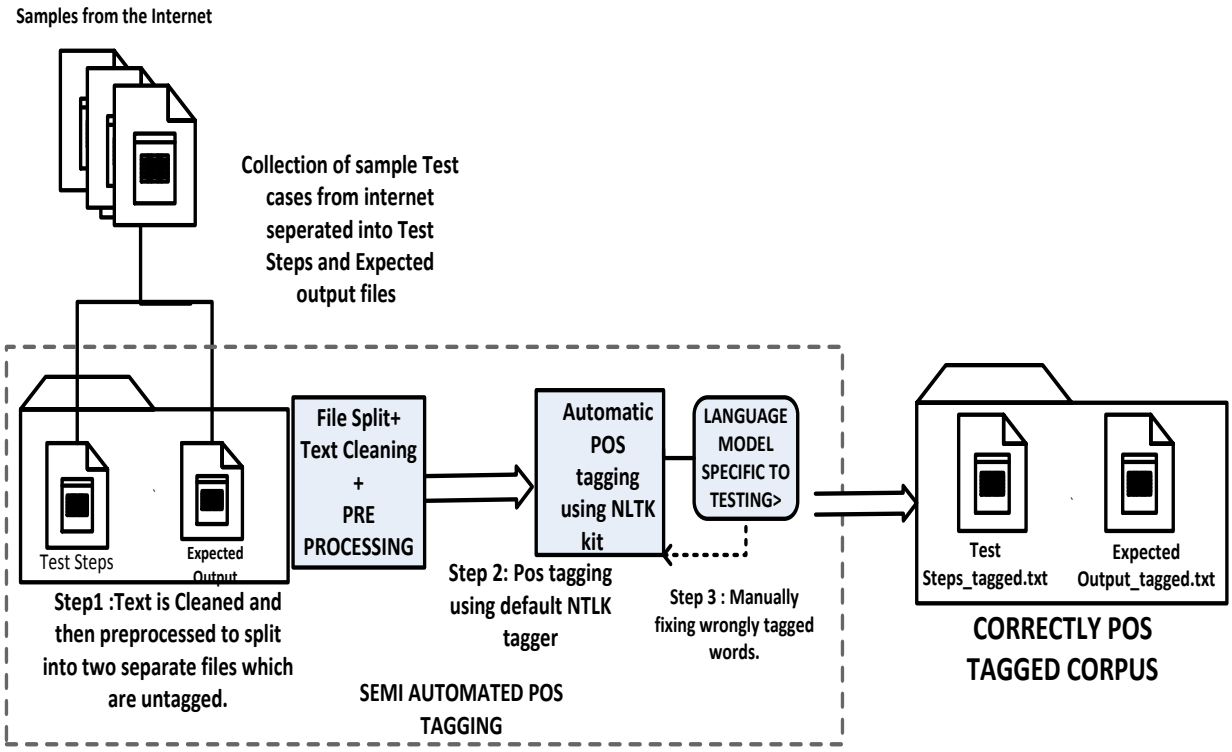

Figure 2:Architecture for Tagged Corpus Creation

3.3.3 Pre-requisites for corpus creation

Step 1:Text cleaning

'Text Cleaning' **i**s **t**he term used in Natural language processing [2] to describe the overall process of converting raw data found on the web into a form useable by NLP algorithms. Following Text cleaning or Preprocessing steps have been done manually to avoid ambiguous test cases in the training data: a) All the text has been converted to lowercase with the help of NLTK toolkit, b) Sentence boundaries were marked with a full stop, c) Wrong misspelt words have been fixed manually, d) Uncommon Abbreviations were expanded, and e) Hyphenated words were split into two words.

Step 2 : Text Clustering

Once the text has been cleaned they were clustered into two categories -Test steps and the Expected Output. This is because the category of words used in writing test steps is different from the words used in expected output and hence the separation. They are segregated as two separate text files and all the test steps are clustered into one file –Test step.txt and their expected output into another-Expected Output.txt (See Figure 2).

Step 3: Semi automatic POS tagging

This component estimates the set of possible tags {T}, for every word in a sentence. We shall call this as Automatic POS tagger module. This module uses NLTK kit to assign parts of speech tags for English words formed in a normal grammatical context initially. The Tag set used for NLTK uses the default Upenn-Tagset [31]. The output of the module would be a tuple set in the (word/tag) format as illustrated below.

Input *: User enters Username and press tab key.*

Output*: User/NN enters/VBZ Password/NN and/CC press/NN tab/NP key/NN ./.*

3.3.4 Testing the Tagged 'Test-Case' corpus (Testing phase)

In the previous sections, we showed how we created the tagged corpus for test cases, which will be used as the training data set for tagging new test cases. Twenty percent of the initial untagged samples collected from the Internet forms our test set. We reserve the rest to validate our tool's processing of new test cases and to verify if it can tag them accurately. For this purpose we have used the Tagged Corpus Reader (as shown below) in NLTK tool

kit. The Tagged Corpus Reader trains itself with the already cleaned, POS Tagged Test Cases and the Expected Output Files.

Reader=TaggedCorpusReader (corpus_root,'Tagged.txt')

With these training sentences, the tagger generates an internal model that has information on how to tag a new word [24].

Back off tagging This is a feature of the NLTK kit that has been used for tagging our entire test input. Back off tagging allows you to chain POS Taggers together so that if one tagger doesn't know how to tag a word, it can pass the word on to the next back off tagger. If that fails it can pass the word on to the next backoff tagger, and so on until there are no backoff taggers left to check.

We tag our sentences at three levels in a chained fashion. A sample code snippet is provided below.

Level1: tagger = TNT Tagger(train_sents)
Level 2: tagger2 = UnigramTagger (train_sents, back off=tagger1)
Level 3:If Level 1 and Level 2 fail, we use the default NLTK tagger

The three level tagging ensures that none of the words are left untagged.

3.4 Phase 2: Creation of Keywords map dictionary

In this phase, we manually create a dictionary of mappings from actions (i.e. verbs) in sample test cases to Selenium keywords, i.e., the "*keyword map*" dictionary. We also create python code that makes a call to the specific Selenium method and link that code to the map information. Later, during automated conversion process, the application can select the appropriate python code when it recognizes a verb in the user written test case.

Thus, this phase is a two-step process:

1. Create mappings from verbs to keywords.
2. Generate code for keywords and then create mappings from each keyword to the appropriate code.

Step 1: Creating mappings from verbs to keywords

After POS tagging, each sentence in a user test case can be processed and represented by the tuple (action, object), where actions are verbs and objects are subjects in the sentence. The actions are manually compared to the list of available Selenium Actions and a file is created with this mapping. File entries have the format *{Selenium keyword: list of user actions that map to the keyword}*. Here is an example of contents of such a file.

{ 'click': 'click' },
{'doubleClick': 'dblclick','doubleclick','double click' ,'click twice' }
{ ' navigate': 'goto','go','get ','get to','navigate'}
{ 'input' : 'sendkeys' ,'enter','type','enter','key in','input'}

In cases where we don't have an obvious map for a particular user action to a Selenium keyword, we use the NLTK kit to find the syntactic distance between those actions to the list of all selenium actions available and return the closest match. This way we are always guaranteed to have a Selenium action (We then manually verify that this match will indeed work as expected). Thus, in the event when the user has used an action 'put' instead of 'Input', the syntactic distance between put and all the Selenium actions ('click', 'doubleclick', 'navigate', 'input') will be calculated and the closest match returned.

Step2: Generating code for keywords and creating mappings from keywords to code

In this step, we create python code that makes a call to specific Selenium methods and map each keyword to the appropriate code. Table 4 below shows how our python class calls Selenium test methods Click and Double Click actions. The class takes actions, objects, and parameters in the constructor of the class and then has methods where keywords are mapped to the appropriate Selenium methods.

Table 2:Sample Selenium Code for Keyword Actions

Sample code for keywords –Click, Double Click	

class SeleniumAction():
  def **__init__**(*self*,*args):
  self.action=args[0]
  self.object=args[1]
  self.parameters=args[2]
// Sample methods below
  def **click**(*self*):
  return *self*.object+*".click()"*
  def **doubleClick**(*self*):
  return self.object+".double_click()"
	The main class -Selenium Action class has a constructor that takes three parameters .<Action, Object, Parameters> as arguments.
	<Click, Text Box,""> will be used to call the click method and the following Selenium Webdriver code generated will be
	"Textbox. Click()"
	<DoubleClick, WebElement,""> will generate String
	"WebElement.double_click()"

3.5 Phase 3: Generation of Test Automation code.

In this phase, the *Autotestbot* system first takes as input the UAT tests written by business users and uses the trained "*test cases corpus*" to POS tag the input tests. An intermediate step that is necessary is to map objects in test cases to physical assets in the code unit under test. This mapping needs to be done only once for a particular user code under test. Next, it applies the *keyword map* dictionary on these POS tagged inputs to convert the test cases to keywords. Finally, the system uses these keywords to create test code fragments, appropriate test setup and teardown methods, and a test driver that can be executed by users.

In this section, we first describe details of the acceptable input format for test cases, the intermediate step of mapping objects in test cases to physical assets, and generation of driver code.

3.5.1 Input format

The input to the system is basically a spreadsheet of test cases written in natural language in a specific format as shown in the table below. In the table, the columns are as follows:

1. *Test Steps*: These are the sequence of actions that the user will take to test the application under test. To work properly, the system needs a complete set of steps in order for each test case.
2. *Expected Output:* This is the expected result from the step.
3. *Parameters:* Some actions require parameters. For example, the "Login" action would require a user login id as a parameter. Some actions do not need any parameters. The current proof of concept tool does not support multiple parameters.
4. *Prior Action:* This is like a pre-condition for the test case. For example, before we logout, we must have logged in.

Table 3:Sample Input spreadsheet format

Test Name	Test Steps	Expected Output	Prior Execution	Parameters
Test Login.	Go to the login screen.			https://mail.google.com/"
	Enter in user id .			username@gmail.com
	Enter wrong password	Gets Message as "Signed In".		password
	Try to click on OK button.			
Test Logout.	Click on Sign Out.	Gets Message as "Signed out".	Test Login.	

3.5.2 Mapping objects

The actual physical assets (for example the objects in the DOM structure of an web application) would be different for each application. After parsing a user test case input spreadsheet and then POS-tagging them, the system generates a new spreadsheet with the tagging information as shown in table 3. Here, the test steps are broken into actions and objects. However, the user must manually map the information for corresponding object links (or Object ids). These are usually Xpath selectors or CSS selectors that uniquely identify the objects in the DOM structure of the Web application. Incorrect object ids will result in failure during execution of test cases as the actions will be invoked on wrong physical objects.

Table 4: Input spreadsheet 2 with Object Id added by user

Test Name	Test Steps				Expected Output	Prior Execution
	Actions	Objects	Object Id	Parameters		
Test Login.	go	login screen	USER ENTERS ID	https://mail.google.com/"},		
	enter	user id	USER ENTERS ID	username@gmail.com		
	enter	password	USER ENTERS ID	password		
	click	ok	USER ENTERS ID		Verify "signed in"	
Test Logout	click	signout	USER ENTERS		Verify "signed out"	Test Login

3.5.3 Generation of Test Automation code

The system will read in the revised spreadsheet (i.e. the one with the objects mapped properly) and then use the keyword mapper module to map each test step to appropriate python code from our code library. A python test file will be created using the format as shown in table 5 that includes necessary import statements, calls to test methods, test setup and teardown etc. Each numbered entry in the table is described here.

1. These are the header files used by the entire Python Selenium Code, which is common for every class. Includes all the header files /import statements that are necessary for execution of the Python unit test class (Pyunit).
2. This shall be the test suite name that extends the Pyunit class.

3. This is test setup for every class that includes initial setup.
4. This is the web driver for Firefox. Our automation can run only on Mozilla Firefox. Running on other browsers can be however a stretch goal in future.
5. This is the web url of the application.
6. These are test methods that are generated for every test case. The number of test methods is directly proportional to the number of test steps.
7. A try-catch is added for every object in case the web driver does not recognize the object.
8. Here, the physical element is found using the Object ids specified by the User.
9. This is just an assertion to check the results of the test case.
10. This is teardown method that is called at the end of every test. It closes the browser for every test case so that a new instance of web driver is launched next time.
11. These are steps for the logout method and are similar to the steps for the login method.

A test case consists of input data that is fed into the application under test, and the expected output for that particular input. The expected output for a user interface test case can be as diverse as a text on the Screen, completion of loading of a page, appearance of a valid element on the page, and opening of a dialog message, to name just a few. However, for simplicity purposes, we have considered only one final expected output per test case.

Table 5:Sample generated Webdriver automation test class file

Generated Test Automation File

```
from selenium.webdriver.common.keys import Keys
import selenium.webdriver.support.ui as ui
import unittest, time, re
from selenium.common.exceptions importNoSuchElementException----------[1]


class SamplePythonOutput(unittest.TestCase): ----------[2]
   def setUp(self): ----------[3]
      self.browser = webdriver.Firefox()----------[4]
      self.browser.implicitly_wait(30)
      self.base_url = https://accounts.google.com/ ----------[5]
      self.verificationErrors = []
      self.accept_next_alert = True


   def Test_Login(self):----------[6]
      browser=self.Browser
      wait = ui.WebDriverWait(driver,10)

      try:----------[7]
        User_Name=wait.until(lambda        driver:browser.find_element_by_xpath('//div(id='username') -----
[8]
      except NoSuchElementException:
         User_Name.send_keys("username@gmail.com")
         assert 0,can't find User_Name

      try:
         Password=wait.until(lambda    driver:browser.find_element_by_xpath('//div(id='passwd'')
      except NoSuchElementException:
         Password.send_keys("password");
         assert 0,can't find Password
     //Expected Output for an assertion for a text element.
      try:
         ExpectedText=wait.until(lambda driver:browser.findtext('Signed In')
         AssertifTextsPresent(ExpectedText) :----------[9]
      except NoSuchElementException:
         assert 0,"Cant find the text specified-Test Fail"

   def Test_Logout(self):
      {   //code generated similar to Test_Login}----------[11]

   def tearDown(self):
      self.browser.quit()----------[10]

if __name__ == "__main__":
unittest.main()
```

A test case is passed only if the expected output is met and until all the test steps have been executed in the test case. A test case that does not meet the expected conditions after step-by-step execution is an expected test failure. Test failures because of objects being failed to be recognized in the DOM structure or dynamically changing DOM are unexpected test failures. Handling these "unexpected test failures" is a typical challenge in UI testing and this is discussed further in detail in our Results chapter.

3.6 Design Issues

3.6.1 Corpus preparation and storage

The foremost design issue is in collecting quality sample test cases. Since this was a proof of concept, our research was not particular about getting clean data. Instead we preprocessed it manually to remove ambiguous sentences or words. This can be a flaw in design as in the time involved in cleaning data manually would be a costly operation and cannot happen in a professional environment with stringent deadlines. One suggested solution to the problem would be is to pass it through auto spell checker/grammar or use syntactic parsing where ambiguous /misspelt words shall be automatically changed to the right ones. This can however not guarantee in effectively correcting all the words.

3.6.2 POS tagging techniques that could have been improvised

The next important design issue is with regards to tagging our test cases. For new unknown words our research tags generically with the basic NLTK grammar kit which can lead to wrong interpretation of the test cases. The use of other methodologies as typed

dependency parsing could have been a better option for tagging new words.

3.6.3 Issues with the Implementation

The biggest challenge during our implementation was finding a suitable web-testing tool. After evaluating few alternatives Selenium was selected as it is the most widely used and has more open source contributors. The fact that we decided to use python for implementation was because NLTK is written in python. With the introduction of Selenium Web driver – python based UI testing is still in development stage and is not popularly used. The improvisation to this design would be is to use text parsers written in java and convert the whole concept to a java based with a compromise on NLTK .

To summarize our approach, we first do some initial setup by building a test cases corpus and a keyword map dictionary. For the particular code under test, we also map objects to physical assets in the code. After these steps, our tool is ready to be used to automatically convert user acceptance tests written in natural language to test code that can be run on the firebox browser.

CHAPTER 4 EVALUATION AND DISCUSSION

In this Chapter we evaluate the usefulness and effectiveness of our approach presented in Chapter 3. We also evaluate against the requirements set in chapter 1 and compare our approach with other relevant tools. Finally, we evaluate the different steps in our process: test case corpus creation, the POS tagging methodologies, and automation code generation.

4.1 Evaluation of our approach

4.1.1 Evaluation against our requirements

The output of our approach is a Selenium Web Driver class file that runs in the browser. Let us try to understand if the code generation actually satisfies the requirements specified in chapter 1.

4.1.2 Test Case Execution

The primary objective of the research was to automate the manually written UAT test case and execute them in the browser. The generated Python class file satisfies this objective. This Python class file runs in the browser with the help of Selenium Web Driver and outputs the result for every test method as a pass or a fail.

4.1.3 Usability for non programmers

Instead of going through the established test case steps manually, the designer is provided with the test source code (the automated test class file that is generated). Natural

language is simple to use and hence a business user can easily write the basic automation with our proof of concept tool.

4.1.4 Reusability

The output file has been generated using our keyword test framework, which was built on the top of selenium Web driver framework. The biggest advantage of generating a source class file in this fashion is that rather than just automating the task in the browser is that - the user can tweak the generated Selenium class file and reuse it to a different set of requirements in future. If the generated test automation does not do the action then modification of the test suite itself is our next step. This can be supported only if the code for the test automation is readily available for the user. This is not available in other frameworks and is one of the strongholds of our research.

4.1.5 Maintainability

Modifying the code is at very minimal level as our tool has already generated the base code. Modification of code can be one or more of the tasks like injecting waits, modifying the object ids, adding try, catch exceptions or even breaking the tests into sub tests. This saves us enormous amount of time in writing code from the scratch .The end user does not need to know every single API of the Selenium Framework and hence profound knowledge in debugging is not required.

4.1.6 Modularity and easy to use

The generated source code a Python class file it satisfies most of the object oriented programming principles. Modularity is maintained in terms of Python Class –Object Method Structure. The entire Excel sheet is put in a test class and every test case becomes a py-unit test method .The test setup is the first method to be called and is executed prior every unit test in the class. Assertions are made for the expected output specified in the expected column of spreadsheet. Every test case simulates the UI action by a user and can be seen executed on the browser. Any test case can be removed or added without affecting other test methods, which makes our approach modularized.

4.1.7 Comparison of existing tools with our approach

Based on the feature requirements discussed above table 6 below evaluates our tool against other relevant tools. The tool is relatively easy to use compared to RTF and Cucumber for the reason free form English test cases are allowed. We realize that the consistency of test case execution seems to be medium compared to other tools. The consistency would definitely improve over time when the corpus size increases. Thus based on the following comparisons we can understand that concept can be extrapolated to have most features required for using it as a framework for writing UAT tests.

Table 6:Comparison of Autotestbot with other tools

Tool	Maintainability	Free form test cases	Reusability	Modularity easy to use.	Consistency test case execution	Complexity for non programmers
Record and Playback					Low	Easy to use
Cucumber	✓		✓	✓	High	Complex
Robot Test Framework (RTF)	✓		✓	✓	High	Medium
Autotestbot	✓	✓	✓	✓	High	Easy to use

4.2 Evaluation of the corpus creation phase

A corpus is made for the study of a particular language. The objective of our Setup Phase was to study the language used by a software tester. Our research did not use human groups or professional testers for writing test cases and therefore the test cases were sampled from the Internet. For research purposes we chose to test a very common scenario that was the login functionality of an email.

In the next two sections we have assessed the corpus creation phase (Phase 1) on the basis of its quality and size. Also we have discussed some factors that could threaten the validity of our training corpus.

4.2.1 Evaluation with respect to corpus quality

With our research we were able to observe that language sentence patterns for a Business User seems to be consistent. Our claim is that a Business user or domain expert can use the same vocabulary [58], used previously in writing test cases. Hence collecting test cases from one particular Business User or a user group can jeopardize the validity of our corpus. It can result in building a biased corpus.

This biased corpus of tagged test cases will not be useful to identify new test cases or tests from a different test user group. Therefore a random sampling from different testers who are unaware of each other's language was adopted in building our corpus. This ensured that new test cases could be tagged.

4.2.2 Evaluation with respect to corpus size

Corpus size influences the quality of research. In "Corpus Linguistics and Technology" [9] the author discusses the different aspects of creating a high quality corpus with emphasis on corpus size. The analogy was with respect to size of Brown Corpus .

In the early years of electronic corpus generation, the Brown Corpus, which contains one million words, was considered to be a standard one. In the Brown Corpus one million words were divided as 7 genres with 500 samples of text and each sample consist of 2000 words. Considering English to be the most common language 1 million words is not a very big corpus but Brown corpus was useful in Information extraction. We can apply a similar approach in building our test corpus from a smaller scale.

Lets us understand how our research in collecting test cases samples can become a powerful knowledge base over time. In a product based organization test cases are written for every software release. Every software release has an addition of features or enhancement of features. The release doesn't go to production until acceptance testing is done. Since User acceptance tests are written for every release, every feature and hence test cases keep growing over time. This serves as a strong source in developing test cases corpus specific to the team.

The usage of words relevant to the product features, usage of product specific actions, action verbs in a UAT test case helps us in building the Knowledge base (KB) for the team. Thus over time if we can build a robust test cases corpus we shall be able to categorize test cases much more effectively. This however can occur over time when the test cases corpus grow in size and the taggers can tag the test cases accurately.

4.2.3 Threats to validity in corpus creation

In this section we shall list the types of issues that can influence the validity of our Test cases corpus.

Conclusion validity

The size of the corpus was one of the main threats in accurately tagging our test cases. The scenario for our research was testing the login functionality of an email application. Our small corpus size was sufficient enough to test this small feature. In order to test the whole email application we need test cases for every feature for the application. On addition of a second feature to be tested, test cases related to that feature have to be added and so on. The

minimum size of the corpus to test an application can be roughly deduced with an equation as follows:

Minimum Corpus size = minimum no Test cases for a feature of the application * no of features

From the above relation, we understand that to collect test cases for all the features of an application can help in complete testing. Creating this kind of an all feature corpus would be one of the biggest challenges in our research.

External validity

There were no threats to validating our research because of external influence. The users on Internet have no dependency or influence with our research. The test cases were randomly collected from the web, which validates that no external factors have influenced our test corpus design.

Construct validity

The following are the factors that affect the construct validity.

- One of the most import threats for using free form test cases was that the samples might actually not be constructed the way we desired for. Let us try to understand this with an example: "Signing in " or "Log in" can vary between applications. For instance the steps involved in "signing in" operation for a Gmail account is not essentially the same in a "Outlook" email application.

- Wrong grammar serves as a potential threat in a wrong understanding of tagged test cases and has to be rectified.
- Short forms /abbreviations have to be looked out for and cleaned.
- Test cases from Non-native speakers of English might have different sentence pattern types as compared to native speaker types. This has to be carefully selected out.

4.3 Evaluation of Phase 2-POS Tagger models

In this section we will evaluate the POS tagging methodologies used in our research. We would try to understand the effect of wrongly POS tagged sentences and how it affects the code generation phase with examples.

4.3.1 POS Tagger model used in our research

The whole process of tagging our test cases was serialized in our research. The TNT taggers does the tagging and pushes to the Unigram Tagger and finally reaches the NLTK default Tagger if the tagging fails at the first and second stage.

TNT taggers – >Unigram Tagger – > NLTK. Default tagger

Trigrams 'n' Tags (TnT) are an efficient statistical part-of-speech tagger. [4] A recent comparison has shown that TnT performs significantly better for the tested corpora. Our training set has been manually tagged and TNT taggers work better since the training set is manually fixed [4]. Our observations say that TNT taggers have been able to identify known test cases better and fail relatively with unknown ones. The following examples will help us illustrate the failure in POS tagger models.

4.3.2 Unknown words usage

In the below example a user writes the test case in natural language as

Enter Username as your hot mail address.

The word “hot” never had a three-word context [36] in our repository and hence Trigram taggers (TNT) fail to identify the test cases .The onus of tagging falls on the next tagger –the unigram tagger. Since the unigram tagger also trained with the same corpus the word ‘hot’ had no single match. Therefore the unigram tagger relies on the last but least –the default NLTK tagger. NLTK tagger successfully tags the word ‘hot’ based on its general usage.

The NLTK default tagger as tags the words below:

E**nter/VB Username/NNP as/IN your/PRP hot/JJ mail/NN address/NN.**

We can see in the above-tagged sentence, ‘hot’ was tagged as an adjective (JJ) and ‘mail ‘was tagged as a common noun (NN), which actually is incorrect. We observe that the above example “hot” and “mail” were considered two words instead of “hot mail “ as a Noun word. So in the above case our taggers have actually failed to identify the unknown words and hence we would be unable to break our sentences in the accurate <Action, Object, parameters> tuple format. This would lead to incorrect keyword-automation code mapping and hence our test cases fail.

4.3.3 Improper breaking of sentences

For instance the user writes the test case as

"double click to select the application app"

Things that can be inferred at the higher level:

a.)'double click' and 'select' are actions.

b.)'application app' is the object.

The test case can be tagged as

Case 1. **double click /VB to/TO select/VB the/DT application/NNP app/NNP**

Case 2. **double/JJ click /VB to/TO select/VB the/DT application/NNP app/NNP**

Our proof of concept tool should do a double click operation on the application app and not a select action (In this case select action can be selecting an area). Also, 'double click' action cannot be pursued for a click action.

The above scenario brings us the fact that our tool should not pursue a wrong mapping of an action and hence an incorrect automation code should not be produced. Incorrect mapping of an action happens because of the improper break up of the sentence. Improper breakup of the sentence is because of wrong tags assigned to the words in the sentence. This results in semantic misinterpretation of the test case actions.

Two similar words with a different context is a problem in our research at the moment. Semantic misinterpretation of test case actions can be prevented at an initial stage by better proof reading. Section 5 gives us some higher-level overview of preventing error prone natural language test cases.

4.4 Evaluation of the Implementation phase

The success of our proof of concept is the quality and accuracy of automation code generated. The approach undertaken right from taking inputs to the user, using the keyword mapper module, the Object recognition capabilities of our Selenium base code and the automation code generator, all determines the quality of our generated class file.

4.4.1 Evaluation of Spreadsheet Inputs

How effective is our design for taking the inputs from the user? The User has to write step definitions for test cases and every row in the test case column corresponds to an executable action. Combination of actions can lead to a much more in depth parsing of text. Hence the spreadsheet format with one action helps us in reducing the parsing overhead. Once the user gives the inputs we parse it, separate the objects ,get the secondary inputs and then maps the object ids. The significance of the secondary input sheet is to ensure every action has only one corresponding web object involved. Every row in the object id column corresponds to only one object, which is an automatically extracted .Our observation claim that the extraction of objects saved a lot of time in identifying Web Objects.

4.4.2 Evaluation of Keyword Mapper Module

As discussed in chapter 3, one motive of our research is to expose the source test class file to the user for reusability purposes. For this purpose every corresponding <action, object > tuple extracted from the natural language text has been generated as python code as a ‘String Buffer” and appended to an executable Python test class file. This has been really the

'most' unique aspect of our keyword based automation framework and the question that arises is about the effectiveness of the keywords to automate the application. These keywords are comprehensive enough to automate the email feature of the application. However at a bigger picture, more keywords have to be added to the Selenium Keyword class file (Refer Table 4) so that keywords not frequently used can be mapped as well.

4.4.3 Evaluation of Selenium framework as our base framework:

There are complex UI scenarios where our concept can fail to automate even if there has been an accurate mapping of keywords. Some of them are discussed below:

Page loading issues

One of the tricky aspects of Selenium based testing is about understanding the loading of a page and when elements appear on pages. All our tests were structured with a pattern to find elements using the find element method of the web driver under a try-catch. However if web driver cannot find the element on the page ,it waits till the time out page period and then throws the "Not found exception". Missing pages with wrong URL parameter also lead to failure of tests.

Visibility of web elements

One of the reasons for the failure of our automation code is because of the visibility of web elements. This happens mostly because the object is in hidden state or the object cannot be identified using the right Path/CSS selectors. Hence our tests have to be tweaked sometimes to get the appropriate results.

Cross browser automation issues

Scripts generated as part of research can work only in Firefox web driver. One of the issues with our Selenium automation can be is X paths used in FF may not work for Internet Explorer or Safari. However, usage of unique selectors can resolve cross browser problems.

Ajax calls

A common problem we faced during automation with selenium web driver was to handle Ajax based pages. Since it is harder to estimate Ajax call completions, Ajax pages automation failed due to timeout issues.

CHAPTER 5 CONCLUSION AND FUTURE WORK

Our objective was to automate applications using natural language scripts. We applied natural language techniques to tag manually written English test cases and map them to automation code. We had to start with creating tagged resources from test cases samples. For this purpose we have built our own test language model to tag test cases and used that later as reference to tag untagged test cases.

The biggest drawback in our research was the corpus size and we tried to achieve the goal with a smaller corpus. So we have worked with methods, so that small amount of tagged resources can be used to effectively carry on the parts of speech tagging task. We extract the actions using NLP techniques and map these keyword actions to the corresponding selenium web driver actions. Though our concept can never guarantee a perfectly automated test suite but it can definitely aid the user in creating one.

5.1 Summary of the research

User acceptance testing depicts the end user satisfaction and hence there is the real need for automating UAT. A lot of researchers have worked in this area since 2005. Our work is most closely related to the Cucumber or Robot Test Framework approaches. However, our research is unique in the following ways:

a. The user is allowed to write test cases in a free form language. Other frameworks force the user to build the test cases using their existing keywords.

b. The tool uses Selenium web driver for its base code generation. This shall be considered as one of the biggest positives of our research as UI automation is popular amongst open source Selenium contributors. Record and play back tools as Selenium IDE gives us the base code but the generated code is never guaranteed to run again after first instance and is difficult to maintain them.

c. The amount of time spent in creating UI automation scripts from scratch is enormous and requires a lot of expertise. Our research gives the user the ability to work on a readily available base code to tune it, as it is nothing other than Selenium Web drive code. Code reuse is the biggest objective of our research. Other frameworks automate using their tool.

5.2 Future Work

During the implementation phase we had come up with several ideas that can further enhance the capabilities of the tool:

1. Make the tool open source similar to Robot Test Framework. More the number of users result in more test cases. When the number of test cases gets added the corpus size increases. As a result, a better categorization of test cases can be achieved if the corpus coverage increases.
2. Develop a mechanism to automatically preprocess input test cases so that clean data is fed to our model.

3. Use a different version of the tool, which uses only the keywords map /automation code library as open source, in situations such as a professional test environment where test cases cannot be shared to outside users.
4. Make the entire concept work on a cloud server like sauce labs [34] where the user does not need to worry about the platform or the type of browsers.
5. Make the concept more abstract by generating automation code as Selenium Page objects [49].

REFERENCES

[1] "All About - User Acceptance Testing (UAT)." Guru99. Accessed June 6, 2014. http://www.guru99.com/user-acceptance-testing.html.

[2] Bajpai, Neha. "A Keyword Driven Framework for Testing Web Applications." International Journal of Advanced Computer Science & Applications 3, no. 3 (2012).

[3] Bc. Dávid Chmurˇciak. "Automation of Regression Testing of Web Applications," 2013.

[4] Brants, Thorsten. "TnT: A Statistical Part-of-Speech Tagger." In Proceedings of the Sixth Conference on Applied Natural Language Processing, 224–31. ANLC '00. Stroudsburg, PA, USA: Association for Computational Linguistics, 2000. doi:10.3115/974147.974178.

[5] Brill, Eric. "A Simple Rule-Based Part of Speech Tagger." In Proceedings of the Workshop on Speech and Natural Language, 112–16. Association for Computational Linguistics, 1992. http://dl.acm.org/citation.cfm?id=1075553.

[6] Burnstein, Ilene. Practical Software Testing: A Process-Oriented Approach. Springer, 2003.

[7] "Cucumber/cucumber." GitHub. Accessed June 7, 2014. https://github.com/cucumber/cucumber.

[8] Corpus Linguistics: An Introduction. Pearson Longman, 2008.

[9] Dash, Niladri Sekhar. Corpus Linguistics and Language Technology: With Reference to Indian Languages. Mittal Publications, 2005.

[10] Diane Mueller. “Survey Says: Selenium by a Nose for Most Popular Test Framework.” Accessed June 8, 2014. http://www.activestate.com/blog/2010/07/survey-says-selenium-nose-most-popular-test-framework.

[11] “Examples for Test Cases in Software Testing?” Answers.com. Accessed June 6, 2014. http://wiki.answers.com/Q/Examples_for_test_cases_in_software_testing.

[12] “FitNesse.UserGuide.FitFramework.” Accessed June 8, 2014. http://fitnesse.org/FitNesse.UserGuide.FitFramework.

[13] Hanna, Milad, Nahla El-Haggar, and Mostafa Sami. “A Review of Scripting Techniques Used in Automated Software Testing.” International Journal of Advanced Computer Science and Applications 5, no. 1 (2014). doi: 10.14569/IJACSA.2014.050128.

[14] “Reducing Testing Effort Using Automation.” International Journal of Computer Applications 81, no. 8 (November 15, 2013): 16–21. doi: 10.5120/14032-1837.

[15] Hartmann, Jean. “30 Years of Regression Testing: Past, Present and Future.” Accessed June 6, 2014. http://www.uploads.pnsqc.org/2012/papers/t-67_Hartmann_paper.pdf.

[16] Hetzel, William C. The Complete Guide to Software Testing. 2nd ed. Wellesley, Mass: QED Information Sciences, 1988.

[17] “How to: Create a Data-Driven Coded UI Test.” Accessed June 8, 2014. http://msdn.microsoft.com/en-us/library/ee624082.aspx.

[18] “http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp#test-Automation-for-Web-Applications,” 2014.

[19] “http://wiki.answers.com/Q/Examples_for_test_cases_in_software_testing], [http://www.testingken.com/forum/showthread.php?t=3076],[vietnamese Board for Testing],” 2010.

[20] Kasurinen, Jussi, Ossi Taipale, and Kari Smolander. “Software Test Automation in Practice: Empirical Observations.” Advances in Software Engineering 2010 (February 4, 2010): e620836. doi:10.1155/2010/620836.

[21] Kent,John ,2007, Test Automation From Record/Playback to Frameworks,Paper given at EuroSTAR 2007, Stockholm

[22] “Keyword-Driven Test Automation Framework | Ranorex Blog.” Accessed June 8, 2014. http://www.ranorex.com/blog/keyword-driven-test-automation-framework.

[23] Kit, Edward. “‘Integrated, Effective Test Desi Gn and Automation.’ Software Development (February),” 1999.

[24] Laukkanen, Pekka. “Data-Driven and Keyword-Driven Test Automation Frameworks.” HELSINKI UNIVERSITY OF TECHNOLOGY, 2006. http://eliga.fi/Thesis-Pekka-Laukkanen.pdf.

[25] Li, Eldon Y. “Software Testing in a System Development Process: A Life Cycle Perspective.” JOURNAL OF SYSTEMS MANAGEMENT, 1990, 23–31.

[26] Lingham, Vinny. “The Growth in Web Appplication Usage in the US.” Vinny Lingham’s Blog. Accessed June 6, 2014. http://www.vinnylingham.com/the-growth-in-web-appplication-usage-in-the-us.html.

[27] "List of Web Testing Tools." Wikipedia, the Free Encyclopedia, June 6, 2014. http://en.wikipedia.org/w/index.php?title=List_of_web_testing_tools&oldid=610400905.

[28] Marcus, Mitchell P., Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a Large Annotated Corpus of English: The Penn Treebank." Compute. Linguist. 19, no. 2 (June 1993): 313–30.

[29] Messer's, Gerard. "Agile Regression Testing Using Record & Playback." In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 353–60. OOPSLA '03. New York, NY, USA: ACM, 2003. doi:10.1145/949344.949442.

[30] Modjeska, Natalia N., Katja Markert, and Malvina Nissim. "Using the Web in Machine Learning for Other-Anaphora Resolution." In Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing, 176–83. EMNLP '03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003. doi:10.3115/1119355.1119378.

[31] Myers, Glenford J. The Art of Software Testing. Business Data Processing, a Wiley Series. New York: Wiley, 1979.

[32] Narayanan Jayaratchagan. "Fit for Analysts and Developers." JavaWorld, June 12, 2006. http://www.javaworld.com/article/2071778/testing-debugging/fit-for-analysts-and-developers.html.

[33] "Natural Language Toolkit." GitHub. Accessed June 6, 2014. https://github.com/nltk.

[34] Pan, Jiantao. 18-849b Dependable Embedded Systems, n.d.

[35] "Paper2 A_Keyword_Driven_Framework_for_Testing_Web_Applications.pdf." Accessed June 8, 2014. http://thesai.org/Downloads/Volume3No3/Paper2-

[36] "Part-of-Speech-Tagging.ppt - Part-of-Speech-Tagging.pdf." Accessed June 8, 2014. http://www.cs.umd.edu/~nau/cmsc421/part-of-speech-tagging.pdf.

[37] Perkins, Jacob. Python Text Processing with NLTK 2.0 Cookbook. Birmingham, U.K: Packt Publishing, 2010.

[38] "Pros and Cons of Keyword Driven Testing." Http://blog.bughuntress.com/automated-Testing/the-Pros-and-Cons-of-Keyword-Driven-Testing, May 3, 2013.

[39] R. Strang. "Data Driven Testing for Client/server Applications." In Software Quality Engineering, n.d.

[40] Rao, Ananth. HP QuickTest Professional WorkShop Series: Level 1 HP Quicktest. Outskirts Press, 2011.

[41] "Robotframework-Seleniumlibrary - A Web Testing Library for Robot Framework - Google Project Hosting." Accessed June 9, 2014. https://code.google.com/p/robotframework-seleniumlibrary/.

[42] Sandipan Dandapat. "Master's Thesis on 'Parts of Speech Tagging for Bengali,'" 2009.

[43] "Sauce Labs: Supported Device, OS, and Browser Platforms." Accessed June 8, 2014. https://saucelabs.com/platforms.

[44] "Semperos/watir-Robot." GitHub. Accessed June 8, 2014. https://github.com/semperos/watir-robot.

[45] Simon Stewart. "The Architecture of Open Source Applications: Selenium WebDriver." Accessed June 8, 2014. http://aosabook.org/en/selenium.html.

[46] Singh, Jyoti, Nisheeth Joshi, and Iti Mathur. "Part of Speech Tagging of Marathi Text Using Trigram Method." International Journal of Advanced Information Technology 3, no. 2 (April 30, 2013): 35–41. doi:10.5121/ijait.2013.3203.

[47] Soeken, Mathias, Robert Wille, and Rolf Drechsler. "Assisted Behavior Driven Development Using Natural Language Processing." In Proceedings of the 50th International Conference on Objects, Models, Components, Patterns, 269–87. TOOLS'12. Berlin, Heidelberg: Springer-Verlag, 2012. doi:10.1007/978-3-642-30561-0_19.

[48] [49] "Step-by-Step Selenium Tests with Page Objects, Dsl and Fun!" Perdido Is Lost! Accessed June 7, 2014. http://luizfar.wordpress.com/2010/09/29/page-objects/.

[49] Thummalapenta, Suresh, Saurabh Sinha, Nimit Singhania, and Satish Chandra. "Automating Test Automation." In Proceedings of the 34th International Conference on Software Engineering, 881–91. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012. http://dl.acm.org/citation.cfm?id=2337223.2337327.

[50] "Untitled - Uat-Test-Process.pdf." Accessed June 8, 2014. http://www.bced.gov.bc.ca/imb/downloads/uat-test-process.pdf.

[51] Vinci Liu and James R. Curran. "Web Text Corpus for Natural Language Processing," NSW 2006sydney.edu.au/engineering/it/~james/pubs/ps/eacl06web.ps.

[52] “Xchmurc_thesis.pdf.” Accessed June 7, 2014. http://is.muni.cz/th/143240/fi_m/xchmurc_thesis.pdf.

[53] “xebia/Xebium.” GitHub. Accessed June 7, 2014. https://github.com/xebia/Xebium.

[54] “Webdriver Introduction — Selenium Documentation.” Accessed June 12, 2014. http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp#test-automation-for-web-applications.

[55] Fewster and Graham, 1999; Kit, 1999; Pettichord, 1999; Nagle, 2000; Zallar, 2001; Rice, 2003

[56] E. Dustin, J. Rashka, and J. Paul. Automated Software Testing. Addison-Wesley,1999.

[57] “Workshop on Controlled Natural Language”, Fuchs, Norbert E. Controlled Natural Language: CNL 2009, Marettimo Island, Italy, June 8-10, 2009, Revised Papers. Springer, 2010.

APPENDIX A1 : DETAILED SOFTWARE REQUIREMENTS

Continued Software Detailed requirements from section 1.3

1. Creation of a POS tagged Custom Test Corpus:

1.1 To collect reasonable amount of Test Samples for a particular functionality to test.(In this case email login functionality)

1.2 Preprocessing of test cases has to be done to correct spell errors.

1.3 Parts of Speech tagging to be done for every test case step and expected output separately.

1.4 Any mismatch of tags to be rectified to create the 'Test-Cases' corpus.

2. Training of the POS tagger:

2.1 To identify and come up with custom tags specific to our 'test cases' corpus.

2.2 Identification of a suitable POS tagger and to train the tagger on the 'test cases corpus"

3. Input test cases from the User.

3.1 Get the input from the user in form of test cases and parse them for Test Steps and Test Output.

3.2 Mapping of Objects and Object IDs from the test cases .

4. **Understanding the Semantics of the Input test cases:**

4.1 Natural language processing for understanding the Semantics of the test case

4.2 Sentence Boundary detection techniques to break combination of test cases.

4.3 Input Test cases to be deduced to <Action, Object, parameter> form,

5. **Natural Language to Automation script conversion.**

5.1 Choice of right automation tool as Selenium, which can run UI, tests in a browser.

5.2 Every deduced <Action, Object, Parameter > should have a corresponding automation script mapped that runs in browser.

6. Assertions for every Test case as fail or pass.

6.1 Every test case in the input given spreadsheet has to be converted to a test method.

6.2 The whole spreadsheet of test cases should be converted to test class.

APPENDIX A2 : VOCABULARY

Black-Box Testing: A type of testing where the internal workings of the system are unknown or ignored. Testing to see if the system does what it is supposed to do.

Capture and Replay : A scripting approach where a test tool records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time. Often also called record and playback.

Functional Testing: Testing to verify and validate the specified functional requirements

Non-Functional Testing: Testing of those requirements other than functional requirements. Stress, performance, compatibility and usability are some examples.

Regression Testing : Retesting previously tested features to ensure that a change or a defect fix has not affected the previous versions.

NLP- (Natural language processing) :Natural language processing (NLP) is a field of computer science, artificial intelligence, and linguistics concerned with the interactions between computers and human (natural) languages.

POS tagging:: Part of speech tagging is the most likely sequence of syntactic categories for set of words in a sentence.

Test Automation: The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes

SUT : System Under Test : Web application under Test.

Test Corpus : a large and structured set of test cases in the form of text files.

NLTK: Natural Language Tool Kit is a set of Python Modules for NLP.

Imperative pattern: English sentence pattern that gives advice or instructions.

Web applications: Applications that can run only in an Internet Browser.

IR: Information Retrieval: Technique to retrieve meaningful information or semantics from a structured text .

Lexical Unit: A single or group of words that form the basic elements of language.

Test Execution: The activity that occurs between developing test scripts and reporting and analyzing test results