DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

PREDICTING RELATED TEST CASE SCENARIOS BY SOURCE CODE CHANGES

by
Onur Cem ŞENEL

December, 2019
İZMİR

PREDICTING RELATED TEST CASE SCENARIOS BY SOURCE CODE CHANGES

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering, Applied Computer Engineering

by
Onur Cem ŞENEL

December, 2019
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"PREDICTING RELATED TEST CASE SCENARIOS BY SOURCE CODE CHANGES"** completed by **ONUR CEM ŞENEL** under supervision of **ASSOC. PROF. DR. MEHMET HİLAL ÖZCANHAN** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Dr. Mehmet Hilal ÖZCANHAN

Supervisor

[illegible]

Doç. Dr. Semih UTKU

(Jury Member)

[illegible]

Prof. Dr. Mehmet Ünlütürk

(Jury Member)

[illegible]

Prof. Dr. Kadriye ERTEKİN

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would first like to express my sincere gratitude to my thesis advisor Assoc. Prof. Dr. Mehmet Hilal ÖZCANHAN of the Graduate School of Natural and Applied Sciences at Dokuz Eylul University. I am gratefully indebted to his very valuable comments on this thesis. Assoc. Prof. Dr. ÖZCANHAN was always ready to support whenever I ran into a difficulty or had a question about my study. He consistently guided me in the right direction with his constructive criticism. Thanks to his resourcefulness, patience, and tolerance, he facilitated the challenges I encountered in this study.

I would also like to acknowledge to dear Mustafa ERŞAHİN who helped and encouraged me during the study. Without his passionate participation and input, this study could not have been successfully conducted.

I owe my special thanks to my dear wife Nazlı ŞENEL for being with me whenever I need her.

Onur Cem ŞENEL

PREDICTING RELATED TEST CASE SCENARIOS BY SOURCE CODE CHANGES

ABSTRACT

Software testing is commonly used for validating software changes, but it is the most expensive phase of the software development life cycle (SDLC). Test case prioritization (TCP) aims to reduce the cost by scheduling the running order of tests to increase the effectiveness of testing; so that most beneficial test cases are executed first, and faults are detected in the early phases of testing. In the study, we present a novel static TCP technique for manual black-box testing. We use a topic modeling (TM) algorithm to extract the functionalities of each test script. This approach allows differentiating and ranking test cases. Cases those test different parts of the system under test (SUT) get higher ranks. Our approach is compared with the manually prioritized test cases of test engineers in the case study of a commercial online banking project. The comparison shows that the average percentage of fault detection (APFD) rates of our approach is higher than the manual prioritization approach.

Keywords: Test case prioritization, topic modeling, black-box testing, software testing

KAYNAK KOD DEĞİŞİMLERİNDEN İLGİLİ TEST SENARYOLARININ BULUNMASI

ÖZ

Yazılımda yapılan değişiklikleri doğrulamak için yaygın bir şekilde yazılım testleri kullanılmaktadır; ancak bu aşama yazılım geliştirme yaşam döngüsü (YGYD) içindeki en maliyetli aşamadır. Test senaryosu önceliklendirme (TSÖ) yöntemlerinin amacı, test senaryolarını testin etkinliğini artıracak şekilde bir sıraya koyarak bu maliyeti düşürmektir. Bu şekilde, en faydalı test senaryoları diğerlerinden daha önce çalıştırılarak hataların test sürecinin erken aşamasında yakalanması sağlanmaktadır. Bu çalışmada, manuel kara kutu testlerinin önceliklendirilmesi için yeni bir statik TSÖ yöntemi öneriyoruz. Her test senaryosunun işlevselliğini bulmak için bir konu modelleme (KM) algoritması kullanmaktayız. Bu yaklaşım test senaryolarını ayrıştırma ve sıralama imkanı sunmaktadır. Test edilen yazılımın (TEY) farklı bölümlerini test eden test senaryoları daha öncelikli olmaktadır. Yöntemimizi ticari bir çevrimiçi bankacılık uygulamasının test mühendisleri tarafından sıralanmış test senaryoları ile kıyasladık. Bulduğumuz sonuçlar, yöntemimizin ortalama hata yakalama yüzdesinin (OHYY) daha yüksek olduğunu göstermiştir.

Anahtar kelimeler: Test senaryosu önceliklendirme, konu modelleme, kara kutu testi, yazılım testi

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE
INTRODUCTION

New versions of application software are released frequently which always include some changes, in the original work. Changing some parts of software could cause new bugs. The testing process of the software development life cycle (SDLC) is used to assure the quality of the software. Every part of the software should be tested before the release of the new version. Automated or manual testing can be used to ensure the software runs correctly. Both of these techniques are time-consuming and costly. Due to the project's budget, there isn't enough time to run all test cases then fix bugs and re-run test cases again. That's why the detection of the failures in the early stages of the testing process is very critical for saving time. Test case prioritization (TCP) is a technique that sorts test cases for execution to reduce testing cost.

In this study, we focus on manual black-box testing. In the black-box testing, there are test case scripts written in natural language. These scripts contain required information (preconditions, steps to perform etc.) to perform a test case. Test engineers perform test cases one by one to test software. Since we are focusing on manual black-box testing, the only data are test case scripts, there are no test and source codes as in the white-box unit testing. Maximizing coverage and diversifying test cases are two common objectives of a TCP technique in the literature. Since we don't have any coverage information of test cases, we choose a diversity-based TCP technique. The motivation behind diversifying test cases is the fact that similar test cases detect the same faults. A diverse set of test cases has the probability of detecting different faults, hence a greater number of faults. To diversify test cases, we use a text mining concept, called topic-modeling (TM). The idea of topic modeling-based TCP is that assign each test case to a topic. If you do not test a topic, you will not detect failures related to that topic. Furthermore, you should choose tests as much as diverse from each other to ensure test different parts of the software. We use Latent Dirichlet allocation (LDA) technique to generate our topic model (Blei, Ng, & Jordan, 2002). We applied LDA on test scripts to extract the topics for each test case. After that, we have topic membership probability vectors for each test case. We use these vectors to calculate the distance between test cases also we choose Manhattan distance as our distance metric. To rank test cases, we calculate distances between all test case pairs and store these distances in a matrix. Then, we add a test case that has a maximum average distance to all others to prioritized tests list. This greedy approach prioritizes all test cases. We evaluate our approach on a commercial online banking project test cases and compare results to manually prioritized test cases by test engineers. The average percentage of fault detection (APFD) rates of our approach is higher than the manual prioritization approach. This higher score shows that faults are detected earlier when TCP technique is used for manual testing. As we mentioned before, it is very important and cost-effective to find faults earlier. According to this reality, TCP reduces testing costs of software significantly.

CHAPTER TWO
SOFTWARE ENGINEERING

2.1 History of Software Engineering

The term software was first used in 1958 by John Wilder Tukey, a statistical expert. Tukey is also defined the *bit* term for binary digits in 1946. Until the second half of the 1960s, software was considered as a part of hardware and a secondary component of computers. The three events that took place in the late 1960s separated the software and hardware industries. The first event is the introduction of the IBM System 360 computers. This gave chance to software companies to develop and sell software for different users. The second event is that IBM announced that they will charge software and hardware separately in 1968. Until that time, customers pay the computer as a whole. IBM says this move is the result of rising software costs. Whatever the reason, IBM's this move enables various software companies to develop IBM-compatible software. The third event is the development of the microcomputer industry. Along with microcomputers, small businesses can also buy and use computers (Campbell-Kelly, 1995).

These improvements in industry led to a rapid growth in computer applications. After a short time, projects have started to fail. Because projects were missing deadlines and over budget. The reason of this failure is there was no proper best practices to develop complex software at scale commercially. They called it the "Software Crisis". It was clear that designing complex software systems would require an engineering discipline (Brooks, 1987).

The term *software engineering* was first used at a conference held by North Atlantic Treaty Organization (NATO) in Garmisch, Germany, in 1968. They had tried to find the best practices to develop a software project by applying the traditional engineering disciplines to software. At the end of the conference, a report was published that defines the foundations of software engineering.

2.2 What is Software Engineering?

There are various definitions of software engineering. We can accept the Naur and Randell's definition as first definition was given at the NATO conference in 1968 (Naur, 1968, p. 136): "Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

Another definition was given in the IEEE Standard Glossary of Software Engineering Terminology ("IEEE 610-1990—IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries," n.d.) is as follows: "Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

Software engineering can be defined using different words. However, the characteristics of the software engineering are always the same. These characteristics distinguish it from programming. Software engineering concerns the large scale and complex programs. It also focuses on efficiency of the development and maintenance. Software engineering has many things in common with other fields of engineering, but it has its own unique methods.

2.3 Software Engineering Phases

Before producing an engine, requirements are analyzed first such as power, efficiency, size, weight etc. Then engineers design the engine taking into account these requirements. Production begins after this design is tested and agreed.

Similar way is followed when developing a software project. Requirements are collected to solve the problem and described clearly. Then software is designed according to these requirements. Development of the software starts after the design is completed. Van Vliet et al. defines software development phases as in Figure 2.1 (van Vliet, 2008).

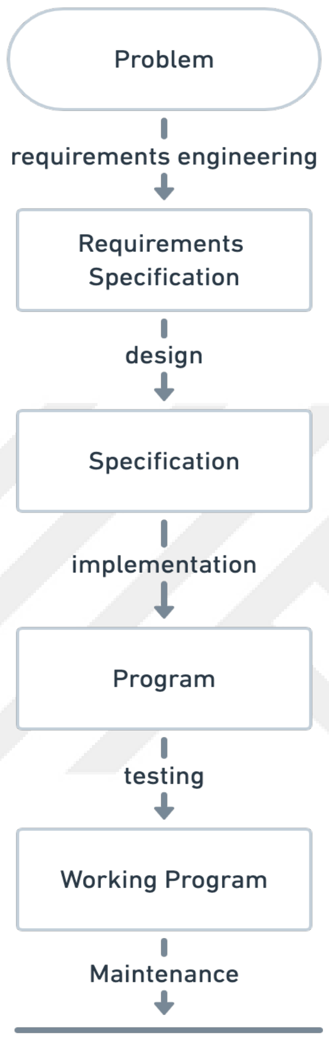


Figure 2.1 A simple view of software development

This a simple model of software development process. It can be slightly changed depending on the size, type and complexity of the project. However, the general process is as given in Figure 2.1.

Requirements engineering is the first step of the development process. During this step, requirements of the software are collected and analyzed. Feasibility study is also part of requirements engineering to specify whether there is a technically feasible solution. At the end of this step, results are collected in a document called requirements specification.

At the design step, whole system is modeled. System is divided into smaller parts called components. The relationship between these components are defined. We also try to separate the *what* from the *how* during the design phase. The results of this step is the technical specification. It is the starting point of the implementation phase.

During the implementation phase, we implement the individual components they defined in the design phase. We got an executable program at the end of this phase.

Testing phase is not a following phase of implementation actually. Testing starts with the requirements engineering and continues. It is refined during the phases. It is cheaper to correct errors if they are detected earlier. Testing is crucial to validate requirements.

Unfortunately, software packages are shipped with errors. Undetected errors during the testing phase should be repaired in the maintenance phase. On the other hand, requirements of the software change during time. These changes are handled during the maintenance phase.

On the early stages of the development, efforts are spent for requirement analysis and they move to implementation on later. But, most of the effort is spent on the testing phase. Efforts of development phases are demonstrated in Figure 2.2 relatively (van Vliet, 2008).

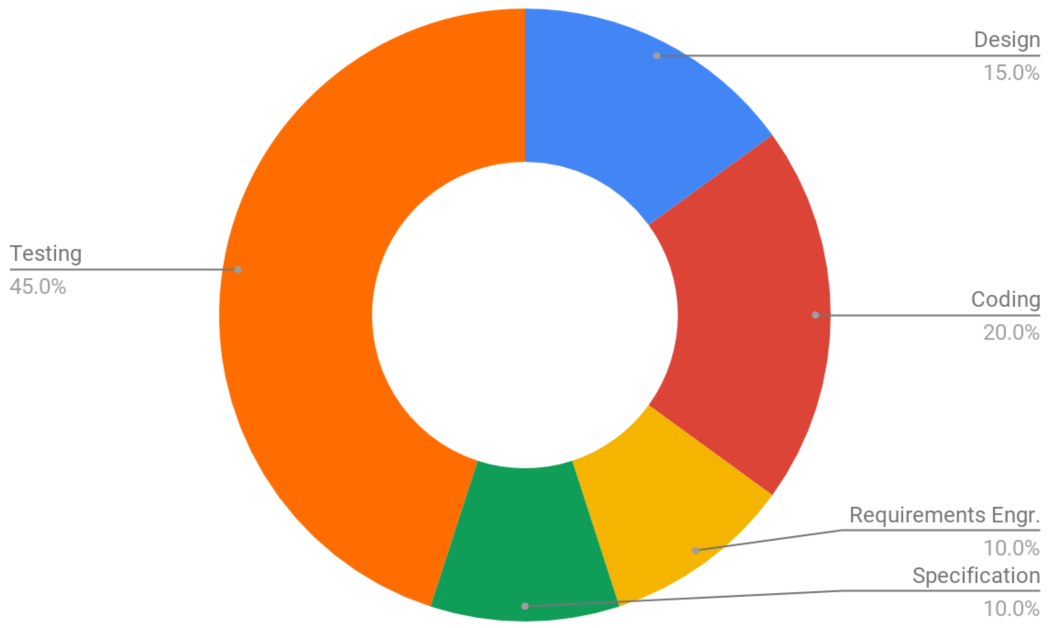


Figure 2.2 Relative efforts of development phases

It is clear that testing phase is one of the most important phases of software development and takes too much effort.

CHAPTER THREE
SOFTWARE QUALITY

3.1 What is Software Quality?

There is no one correct definition of quality. It is really context dependent and a complex concept. The first known definition of quality is made by Shewhart in the beginning of the 20th century as follows (Shewhart, 1930, p. 364):

> There are two common aspects of quality: one of them has to do with the consideration of the quality of a thing as an objective reality independent of the existence of man. The other has to do with what we think, feel or sense as a result of the objective reality. In other words, there is a subjective side of quality.

Kitchenham and Pfleeger published an article about software quality in 1996 (Pfleeger & Kitchenham, 1996). They applied David Garvin's five views of product quality to software quality:

- Transcendental View: This perspective sees quality as something that can be recognized but hard to define.
- User View: This view sees quality as fitness for purpose.
- Manufacturing View: This perspective represents quality as conformance to specifications.
- Product View: In this view, quality perceived as tied to inherent characteristics of the product.
- Value-based View: This perspective implies quality as dependent on the amount a customer is willing to pay for it.

3.2 Software Quality Models

Several software quality models have been proposed over the years. The aim of these models is to define quality and its attributes. The first model is defined by McCall in 1977 (McCall, Richards, & Walters, 1977). This model is developed for large projects in the United States military. McCall defined 55 quality attributes firstly and called them *"quality factors"*. Then McCall reduced the number of quality factors to 11 for the simplicity. These 11 factors are grouped into three categories as product operation, product revision, and product transition factors.

- Product Operation: Correctness, Reliability, Efficiency, Integrity, Usability
- Product Revision: Maintainability, Flexibility, Testability
- Product Transition: Portability, Reusability, Interoperability

A well-known triangle summarizes McCall's quality model under three perspectives which are defined above. Figure 3.1 shows McCall's triangle of quality (Cavano & Mccall, 1978).

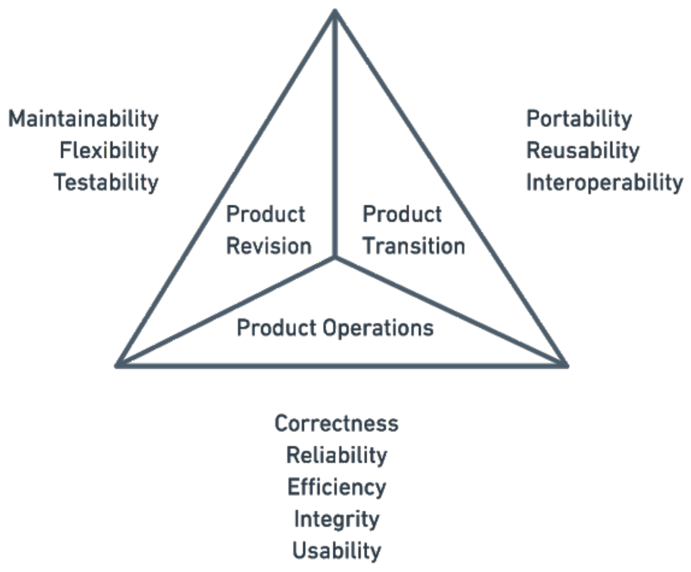


Figure 3.1 McCall's triangle of quality

In 1978, Boëhm has extended McCall's quality factors by adding nine new attributes (Boehm, 1978). Table 3.1 shows the list of both models.

Table 3.1 Software quality models

McCall	**Boëhm**
Correctness	Clarity
Reliability	Modifiability
Efficiency	Modularity
Integrity	Documentation
Usability	Resilience
Maintainability	Understandability
Flexibility	Validity
Testability	Generality
Portability	Economy
Reusability	Correctness
Interoperability	Reliability
	Efficiency
	Integrity
	Usability
	Maintainability
	Flexibility
	Testability
	Portability
	Reusability
	Interoperability

There are some other software quality models proposed by international institutions. International Organization for Standardization (ISO) developed the ISO 9126 quality model. This model contains six quality attributes: efficiency, maintainability, reliability, functionality, usability, and portability. The other model is Capability Maturity Model (CMM) was developed by the Software Engineering

Institute (SEI). CMM model presents five levels of organizational “maturity” that determine effectiveness in delivering quality software.

3.3 Why Software Quality is Important?

Software plays an important role in our lives today. We use various software in a day, and we want to use this software easily and without getting any errors as a user. The quality of some software may even affect the human life and safety. For example, self-driving cars software or auto-pilot systems on planes.

On the other hand, poor quality software costs money to software development companies, during the maintenance period. Fixing these issues cause some delays on the release of the software. Customer satisfaction is also affected by the buggy software.

CHAPTER FOUR
SOFTWARE TESTING

Quality of a software is very critical for the success of that software project as described in the previous chapter. Software testing is an important process to achieve this quality and to ensure that the software is working as expected. Software testing also helps to find errors before releasing the software. We can summarize the software testing processes in three steps such as test, finding faults, fixing faults. Testing process can be manual or automated. There are many types of software testing such as unit testing, regression testing, user acceptance testing, load test and so on (Naik & Tripathy, 2011).

4.1 Basic Software Testing Concepts

4.1.1 Verification and Validation

Verification and validation are two similar concepts to each other. They are frequently used concepts in software testing.

- Verification: Verification activity allows us to determine whether the software product satisfies the requirements. That product does not need to be a final product, it can be an intermediate product in the development phase.
- Validation: Validation activity focuses on the final product. This activity helps us to confirm that software product satisfies customer's expectations.

4.1.2 Error, Fault, Failure and Defect

The four concepts are related but there are major differences between them.

- Error: It can be defined as a *"state"* of the software. In the case of it is not handled correctly, it can cause a failure.
- Fault: A fault is the cause of an error.

- Failure: It occurs when the observed behavior of the software differs from the expected behavior.
- Defect: It is a very close concept to fault and generally used as a synonym of fault. The *"bug"* term is also used.

4.2 Objectives of Testing

There are different stakeholders in the testing activity. These stakeholders are developers, test engineers, business analysts, project managers and customers. The testing process has different meaning for each stakeholder since they view the testing process from different views.

- It does work: In the development phase, developers want to test a unit or entire software is working correctly or not. Because of psychological reasons, the aim of testing here is to show that it works.
- It does not work: After some level of success is achieved, developers try to find some faults with more tests. The goal is here to try to make a unit or entire software fail.
- Reduce the risk of failure: Performing more tests decreases the failure rate of software. Therefore, an acceptable level of failure rate can be achieved.
- Reduce the cost of testing: Decreasing the number of tests is not an acceptable solution to reduce the cost of testing. Selecting effective test cases and performing order of test cases is important to reduce the cost.

4.3 Testing Methods

Black-box, white-box and grey-box testing are three common methods for software testing.

4.3.1 Black-box Testing

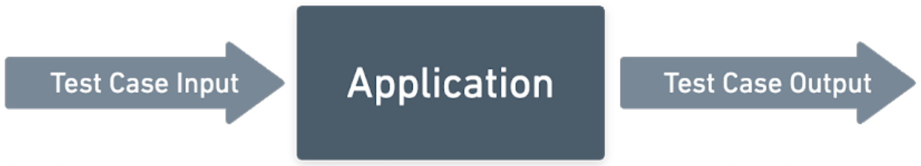


Figure 4.1 Black-box testing

Black-box testing method focuses on output of the system with given inputs. Whole system considered as a black box as shown in Figure 4.1. Testers does not know the internal structure of the system; they work at user interface level. Testers check the output of system according to the requirement specifications. It makes sure that input is properly processed and output is correctly produced (Khan, 2010). This type of testing is sometimes called as functional testing. Advantages and disadvantages of black-box testing are explained below (Sawant, Bari, & Chawan, 2012).

4.3.1.1 Advantages of Black-box Testing

- The number of test cases are reduced to achieve reasonable testing.
- The knowledge of internal structure of the system is not required.
- Test engineer and developer both are independent of each other.
- More effective on larger units of code than white-box testing.

4.3.1.2 Disadvantages of Black-box Testing

- Specifications should be clear to design test cases correctly.
- Some parts of the code are not tested.
- Chances of having unidentified paths during this type of testing.
- Chances of having repetition of tests that are already done by the developer.

4.3.2 White-box Testing

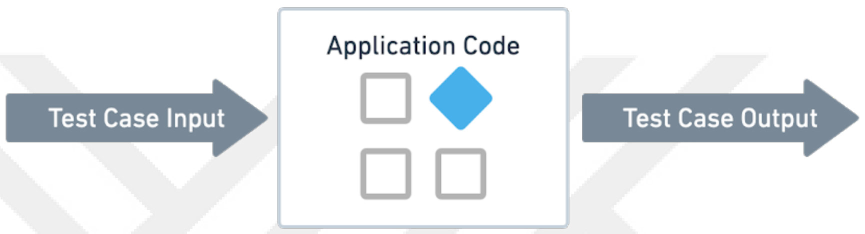


Figure 4.2 White-box testing

White-box testing method is contrasted with black-box testing method. Testers, generally developers, know the internal structure of the system as shown in Figure 4.2. White-box testing is highly effective in detecting and fixing faults, because faults can often be found before they cause trouble (Jovanović, 2006). Tests are written in source code level. The aim of this type of testing is to check data and control flows. It is the process of giving the input to the system and checking how the system processes that input to generate the desired output. Glass box and structure-based testing names are also used for white-box testing. Advantages and disadvantages of white-box testing are explained below (Sawant et al., 2012).

4.3.2.1 Advantages of White-box Testing

- All logical decisions and independent paths in a module are tested.
- Internal data structures are tested to maintain their validity.
- Hidden errors in the code are revealed.
- The reason of the fault can be easily detected by the developers.

4.3.2.2 Disadvantages of White-box Testing

- The cases omitted in the code are missed out.
- An experienced tester is required to perform this type of testing since the knowledge of internal structure of the system is a prerequisite.
- Testing every part of the code is nearly impossible.

4.3.3 Grey-box Testing

Grey-box testing method is a combination of black-box and white-box testing methods as shown in Figure 4.3. This method is used to test a part of software system using some knowledge of the internal structure. The information of the internal structure in the grey-box testing is more than black-box testing, but less than white-box testing (Khan, 2010).

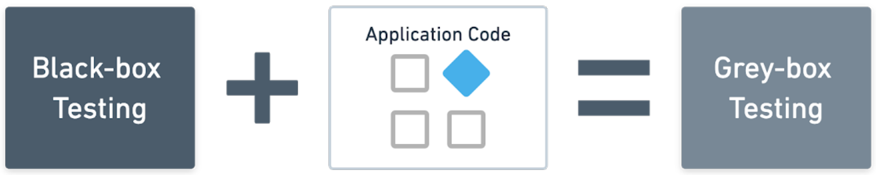


Figure 4.3 Grey-box testing

4.4 Testing Levels

There are different software testing levels for each stage of the software development life cycle. A software system goes through four stages of software testing before it is released. These four levels are unit, integration, system, and acceptance level testing. The first three levels of software testing are performed by different stakeholders in the development team, whereas the acceptance testing is performed by the customers. These four levels of software testing process can be shown in classical V-model in Figure 4.4.

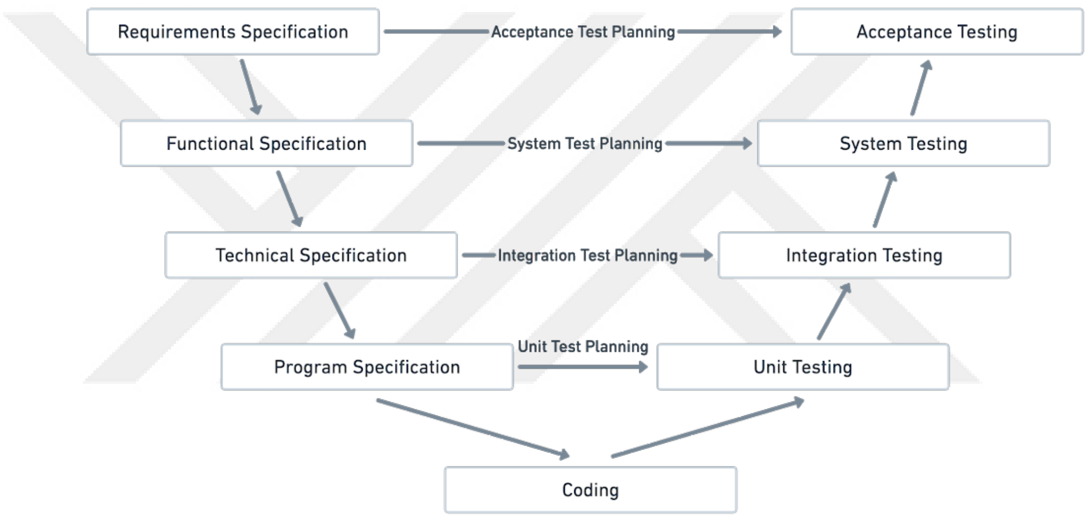


Figure 4.4 Development and testing phases in V-model

In the V-model, there is a corresponding testing phase for every software development life cycle phase. This is a highly disciplined model and the next phase starts only after completion of the previous phase.

It is named as V-model since the entire figure looks like a "V". The V-model is also known as Verification and Validation model. Activities in the left side of the model are software development activities in the SDLC. These are verification phases. Activities in the right side of the model are software testing activities in the software testing life cycle (STLC). These are validation phases. The coding phase in the center bottom of the model joins two sides of the model (Naik & Tripathy, 2011).

Verification phases in the V-model are explained in the Chapter 2 under the SDLC section. Validation phases are explained in detail below.

4.4.1 Unit Testing

This is the first level of software testing. Unit testing is generally performed by the developers. Developers test the small parts of the software such as components, functions, classes. They want to ensure that the parts work as they were designed to. Each unit test works individually.

4.4.2 Integration Testing

Integration testing is used to validate two or more integrated units work together as expected. These tests are often based on user scenarios. Unit tests can be implemented by developers or test engineers.

4.4.3 System Testing

System testing is a black-box testing method used to ensure the whole system meets specified requirements. It includes various tests such as functionality, performance, stress, security and load testing. System testing is generally performed by test engineers before the release of the software.

4.4.4 Acceptance Testing

This is the final level of software testing. Acceptance testing is used to ensure that the software is ready for production. It is performed by the product owner to find out if the software meets all requirements.

CHAPTER FIVE
TEST CASE PRIORITIZATION

As explained in the previous chapters, software quality is very important for the success of the software project. We use software testing process to achieve this quality, but it is an expensive process in the software development. Therefore, effectiveness of the testing process plays an important role.

In large-scale software development, performing all test cases may take days or weeks. Detecting faults as early as possible, e.g., on first day rather than last day, increases the effectiveness of the testing process. Thus, developers have more time to find and fix faults. Early detection of faults depends on the performing order of test cases. All test cases should be ordered by their priority before the execution to detect possible faults with fever test cases. This is the TCP approach.

There are different TCP techniques in the literature. Details of these techniques are explained in the following section.

5.1 TCP Techniques

Khatibsyarbini, Isa, Jawawi and Tumeng reviewed 80 TCP studies published between 1999 and 2016. The authors have selected 19 studies to determine the basic flow of TCP process. In these studies, TCP processes were clearly stated. They illustrated the basic flow of TCP process based on these 19 studies as shown in Figure 5.1.

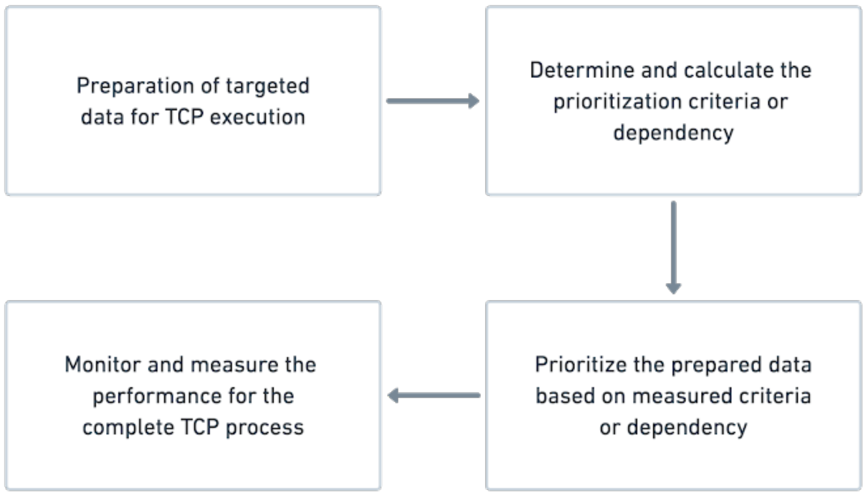

Preparation of targeted data for TCP execution
Determine and calculate the prioritization criteria or dependency
Prioritize the prepared data based on measured criteria or dependency
Monitor and measure the performance for the complete TCP process

Figure 5.1 Standard flow of a TCP technique

As shown in Figure 5.1, TCP process starts with the preparation of targeted data. This data can be specification models, execution information, source code, test cases etc. The second step is determining and calculating prioritization criteria or dependency based on the data chosen. This criterion can be coverage information of the test cases in a coverage-based TCP approach. The next step is prioritization process. After the prioritization step, monitor the results and measure the performance of the approach using a metric.

Khatibsyarbini, Isa, Jawawi and Tumeng also specified the evaluation metrics used in TCP studies. They found that 5 different evaluation metrics are used to measure the performance of a TCP approach. Figure 5.2 shows that the most widely used metric is APFD with a 51% distribution, followed by Coverage Effectiveness (CE) 10%, APFDc (APFD with cost consideration) 9%, time execution 7%, and others 23% (Khatibsyarbini, Isa, Jawawi, & Tumeng, 2018).

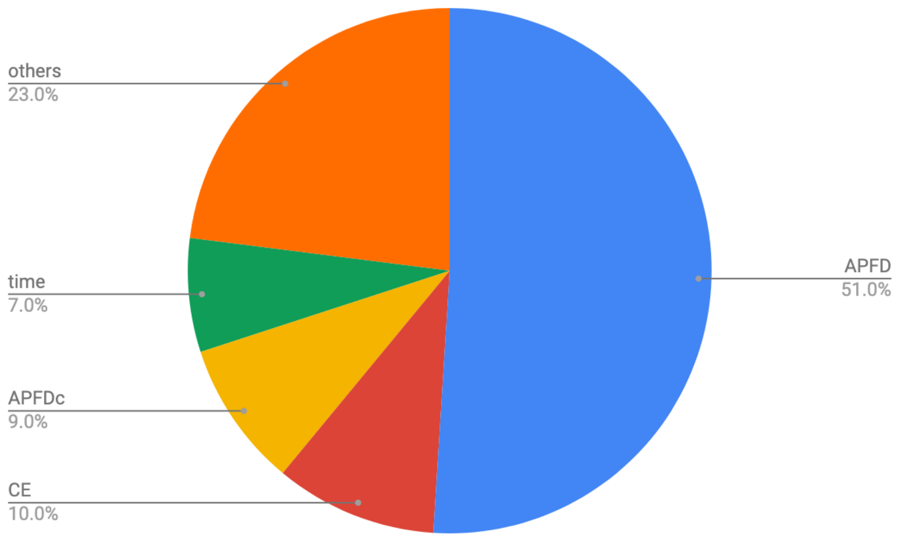


Figure 5.2 TCP evaluation metrics type

In general, there are five categories of TCP techniques based on the available data: white-box execution-based, black-box execution-based, grey-box model-based, white-box static and black-box static prioritization. In addition to these categories, there are two different maximization strategies can be used in TCP: maximizing coverage and diversifying test cases.

In execution-based approaches, the execution information of the test cases is required. Some of the previous studies prioritize test cases by maximizing source code coverage. Statement-level execution information is mostly used as the input for prioritization. Wong et al. add source code change information between versions to the prioritization process to assign high priority to test cases those probably related to modified parts of the source code (Wong, Horgan, London, & Agrawal, 1997). Other studies use diversification to prioritize test cases. Simao et al. use a neural network to find the most dissimilar test cases (Silva Simao, De Mello, & Senger, 2006). Yoo et al. differentiate test cases' execution profiles by a clustering algorithm (Yoo, Harman, Tonella, & Susi, 2009). Execution-based based prioritization studies are generally in the white-box category. Sampath et al. present a black-box execution-based TCP technique (Sampath, Bryce, Viswanath, Kandimalla, & Koru, 2008). They use user activity logs as the execution information in web applications. Accessing the source code is not required by using these logs.

The specification models of the source code and test cases (e.g. UML state diagrams) is used in model-based TCP approaches. Since the source codes of the system under test (SUT) are not required in these techniques, all model-based TCP techniques are gray-box. Execution information may not be available in some projects therefore model-based approaches can be used in these projects. Hemmati et al. and Korel et al. proposed model-based TCP approaches using specification models of the source code (Hemmati, Arcuri, & Briand, 2013; Korel, Koutsogiannakis, & Tahat, 2007). Korel et al. strive to achieve maximum coverage of the model by calculating the difference of the model between two versions (Korel et al., 2007). Hemmati et al. use diversity-based algorithms to calculate the similarity between test cases' paths in the state model then prioritize test cases by their paths similarity (Hemmati et al., 2013).

In a static prioritization approach, execution information and specification models are not required. The source code of the SUT and test cases or test scripts should be available. Zhang et al. propose a call graph-based technique as a white-box static prioritization (Zhang, Zhou, Hao, Zhang, & Mei, 2009). This a coverage-based prioritization approach that works on the static call graph of test cases. A greedy algorithm maximizes the number of source code functions those are covered by the test case to prioritize test cases. Ledru et al. propose another technique that is string-based black-box static TCP technique (Ledru, Petrenko, Boroday, & Mandran, 2012). They measure the distance between test cases to identify their similarity. They compare a number of distance metrics such as Manhattan, Euclidean, Hamming and Levenshtein. Their study shows that the Manhattan distance is the best metric in the context of average performance for fault detection. Ledru et al. also choose a greedy algorithm to maximize diversity between test cases.

There are a few studies that only use test scripts as the input of the prioritization process. Hemmati et al. interested in the TCP problem in the context of manual system-level black-box testing which is our focus in this paper also (Hemmati, Fang, Mäntylä, & Adams, 2017). They use test scripts written in natural language (e.g. instructions in

English) from the Mozilla Firefox projects. They implement several existing TCP techniques and adapt them to the domain of black-box system-level test prioritization. They use the APFD metric for comparing the effectiveness of these TCP techniques.

CHAPTER SIX
PROPOSED TOPIC-BASED TCP TECHNIQUE

6.1 Motivation Behind Proposing a New Technique

In this study, we focus on black-box static prioritization in the context of manual testing. There are very limited studies that only use test scripts written in natural language to prioritize test cases. Most of the current studies focus on prioritization of automated test cases such as unit tests. Whereas, manual testing is the most common testing technique in the industry today. We implement a topic-based TCP technique to prioritize manual black-box test cases. We use test scripts from a commercial online banking project which is being developed with Agile methodologies. We will give detailed information about our test results in Experimental Results section.

6.2 Differences from Other Studies

The proposed approach in the thesis uses test case scripts. To the best of our knowledge, there are very limited number of studies on test case prioritization in the literature. Most of the former studies use the source code of the developed software. It makes our approach more flexible. Any test case script written in natural language can be given as input. It does not depend on any programming language syntax such as Java unit test source codes.

We designed our tool as modular, and it can be easily integrated into continuous integration systems of software development companies.

6.3 Technical Details of Proposed TCP Technique

6.3.1 Topic Modeling

The data which can be collected from various resources is growing continuously in recent years. Most of these data is unstructured. It is hard to obtain useful information from such data. Today, we have new and powerful techniques to extract valuable information from a large amount of data with the help of technology. One of these methods is text mining.

Topic modeling (TM) is a text mining method to abstract textual data from any text document collection. TM is different from other text mining approaches like rule-based ones. It gets the topics which are presented in text data. TM is also categorized in the unsupervised machine learning approaches.

Topics are a group of words from document collection. "A repeating pattern of co-occurring terms in a corpus" definition is used for topic. Topics summarize large text collections. TM is very useful for categorizing large amount of unstructured data. This unstructured data can be social media posts, emails and test scripts in our work.

The topics and topic assignments are illustrated in Figure 6.1 (Blei, 2012).

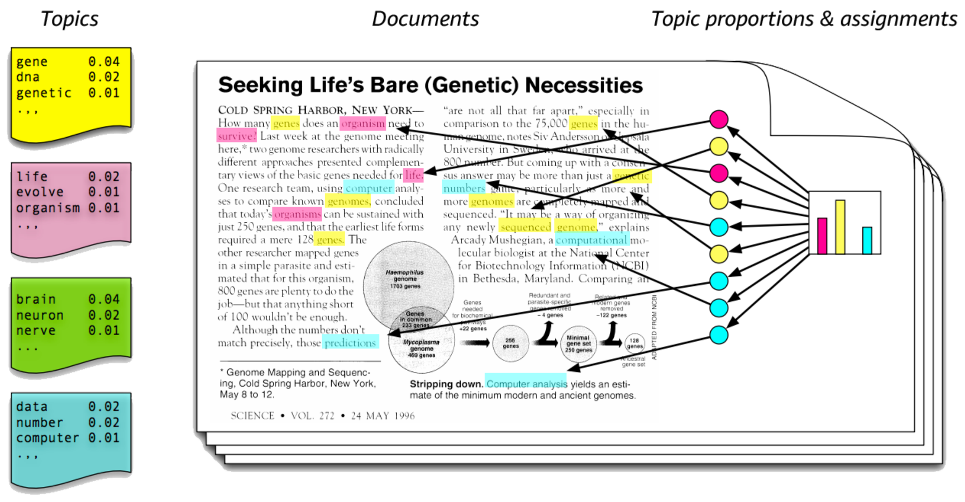
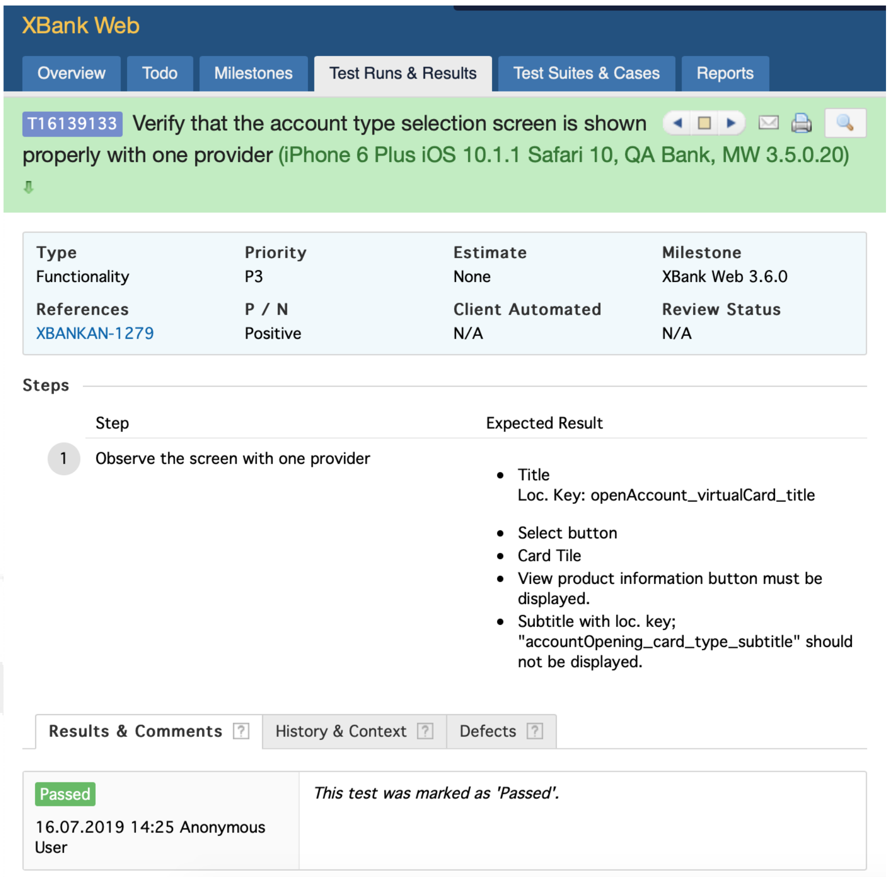


Figure 6.1 Topics from an example text

In using a topic-based TCP approach, our aim is to prioritize test cases effectively and help test engineers to save time when running the tests of a project. Since TM does not require any training data, it can be applied to test cases of any project, quickly. In addition, it is fast enough to handle numerous documents.

6.3.1.1 Data Preprocessing

Preprocessing is one of the important processes in text-related machine learning studies such as text mining, information retrieval etc. As mentioned before, topic modeling is categorized in unsupervised machine learning methods. Therefore, preprocessing is the first step of the proposed approach to have better experimental results. Unnecessary characters such as punctuation, spaces, stop words should be removed from the text data. Stemming is also required to improve IR performance. Words are reduced to their word stems with stemming. Stemming approaches depend on the structure of the language and is not a concept applicable to all languages. We have suffixes like "-ing" and "-ed" in the English language. It is useful to cut off these suffixes to map the words to their stems. For example, "work", "working" and "worked" words will be converted to the same stem of "work" after stemming ("Snowball: A language for stemming algorithms," n.d.).



XBank Web

Overview | Todo | Milestones | Test Runs & Results | Test Suites & Cases | Reports

T16139133 Verify that the account type selection screen is shown properly with one provider (iPhone 6 Plus iOS 10.1.1 Safari 10, QA Bank, MW 3.5.0.20)

Type	Priority	Estimate	Milestone
Functionality	P3	None	XBank Web 3.6.0
References	**P / N**	**Client Automated**	**Review Status**
XBANKAN-1279	Positive	N/A	N/A

Steps

	Step	Expected Result
1	Observe the screen with one provider	• Title Loc. Key: openAccount_virtualCard_title • Select button • Card Tile • View product information button must be displayed. • Subtitle with loc. key; "accountOpening_card_type_subtitle" should not be displayed.

Results & Comments | History & Context | Defects

Passed 16.07.2019 14:25 Anonymous User	*This test was marked as 'Passed'.*

Figure 6.2 A sample test case script from our experiment

As explained above, the proposed approach in the thesis uses test case scripts rather than test case source codes. Figure 6.2 shows the sample test case script of our experiment which is testing an account type selection screen in an online banking web application. The details of the test case can be shown in this screenshot. There is a title of test case in the green section. Other details such as type, priority and milestone follow it. Steps section contains an ordered list of steps to perform this test case and the expected result of each step. Finally, there is a status of the test case at the bottom.

The test case scripts usually explain the test steps to perform in natural language (e.g. instructions in English). The primary step of the data preprocessing is applying some preprocessing steps to the text of test cases. First, we remove special characters (e.g., “-”, “&”, “?”, “!”) and common English stop words. Then we stem each word into its base form. Finally, we convert each word to lowercase. Our proposed method uses the preprocessed text as input data to apply the TM algorithm for extracting topics. Figure 6.3 shows an example test case title before and after preprocessing step from our case study. The text in the first box is the original title of a test case script. The text in the second box is the preprocessed status of the original title. As a result of stemming, some words are converted to their stem. For example, “activation” is mapped to “activ”. “with”, “on” and “the” words are removed because they are stop words. Special characters, “-”, “/”, are also removed.

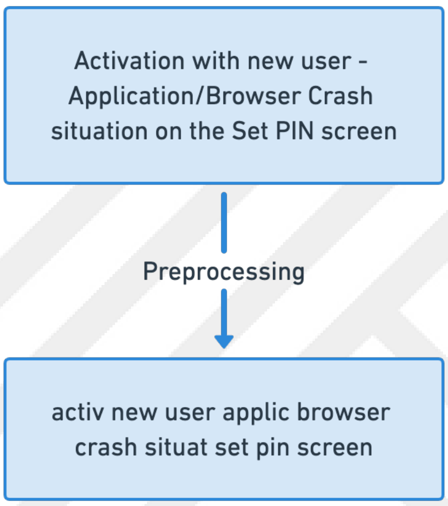

Activation with new user - Application/Browser Crash situation on the Set PIN screen
Preprocessing
activ new user applic browser crash situat set pin screen

Figure 6.3 An example text preprocessing

We use NLTK package of Python to perform these preprocessing tasks (“Natural Language Toolkit—NLTK 3.4.5 documentation,” n.d.). And, we choose PorterStemmer as our stemming tool (Porter, 2006).

6.3.1.2 Latent Dirichlet Allocation (LDA)

There are many TM techniques to obtain topics from corpus: Term Frequency, Inverse Document Frequency and LDA.

We prefer the most commonly used LDA model in natural language processing. And, we use Gensim package of Python programming language as our LDA implementation tool (Řehůřek & Sojka, 2011).

LDA views documents as a mixture of topics and views topics as a mixture of words. LDA represents the corpus as a document-term matrix. Table 6.1 shows a document-term matrix of *n* documents and *m* words. $D_1, D_2, \ldots, D_n$'s are documents and $W_1, W_2, \ldots, W_m$'s are words in the corpus. The number in each *i*, *j* cell shows the frequency of word *Wj* in document *Di*.

Table 6.1 LDA document-term matrix

	W_1	W_2	W_3	W_m
D_1	0	2	1	3
D_2	1	4	0	0
D_3	0	2	3	1
D_n	1	1	3	0

LDA creates two lower dimensional matrices from this document-term matrix. Table 6.2 shows these matrices. The first matrix is the document-topics matrix where *k* is the number of topics. The second matrix is the topic-terms matrix where *m* is the number of words.

Table 6.2 LDA lower dimensional matrices

	K_1	K_2	K_3	K_k
D_1	1	0	0	1
D_2	1	1	0	0
D_3	1	0	0	1
D_n	1	0	1	0

	W_1	W_2	W_3	W_m
K_1	0	1	1	1
K_2	1	1	1	0
K_3	1	0	0	1
K_k	1	1	0	0

These two matrices provide document-topic and topic-word distributions. The main aim of LDA is improving these distributions. The algorithm iterates through each word for each document and it makes a new topic-word assignment to adjust current assignment. A new topic is assigned to a word with a probability. There are two probabilities they are calculated for every topic. The first one is the rate of words in the document that are assigned to a topic currently. The second one is the rate of assignments to a topic over all documents that come from this word. The product of these two probabilities is used to update current topic-word assignment. Document-topic and topic-word distributions will be reasonably good after a number of iterations.

LDA needs the number of topic parameter, *K*. Choosing the optimal value of *K* is another research decision matter. Larger *K* values produce finer-grained topics and smaller *K* values produce coarser-grained topics. Previous studies in software engineering, commonly used *K* values ranging from 5 to 500 (Griffiths, Steyvers, & Tenenbaum, 2007). Thomas et al. use $K = N/2.5$ to produce medium-grained topics, where *N* is the number of documents in the SUT (Thomas, Hemmati, Hassan, & Blostein, 2014). Similarly, we choose *K* as $N/2.5$ rounding to the nearest integer.

There are other parameters of LDA such as alpha, beta, number of topics terms and number of iterations. We use the default values in Gensim package for other parameters of LDA.

There is a small example from our case study project to show how LDA works. For simplicity, we choose 3 test case titles as documents. These 3 documents can be seen in Table 6.3, preprocessed states are also presented in Table 6.4.

Table 6.3 Example documents before preprocessing from case study

Doc ID	Text
D_1	Displaying account transactions
D_2	Displaying Account Transactions if there is no transaction to display
D_3	Service error controls on Account List & Transactions

Table 6.4 Example documents after preprocessing from case study

Doc ID	Text
D_1	display account transact
D_2	display account transact transact display
D_3	servic error control account list transact

After preprocessing, we have 7 unique tokens (words): “account”, “display”, “transact”, “control”, “error”, “list”, “servic”.

Since we have 3 documents calculated topic count is equal to 2 by using $K = N/2.5$ formula. LDA generates following topic-term matrix in Table 6.5.

Table 6.5 An example topic-term matrix from case study

	control	servic	list	error	account	transact	display
K_1	0.159	0.159	0.159	0.159	0.157	0.152	0.054
K_2	0.044	0.044	0.044	0.044	0.217	0.307	0.301

Document-topic probabilities are presented in Table 6.6.

Table 6.6 An example document-topic matrix from case study

	K_1	K_2
D_1	0.14174	0.85826
D_2	0.0924721	0.9075279
D_3	0.8954705	0.10452951

6.3.2 Distance Calculation

After applying LDA, we have a document-topic matrix that represents the topic membership probability of each test case. We need to calculate distances between each test case using their topic membership probability values.

There are different techniques to calculate the distance between two points in mathematics. The most known techniques are Euclidean and Manhattan. Ledru et al. found that the Manhattan distance metric is optimal for string-based TCP, therefore, we use the same metric (Ledru et al., 2012).

Manhattan distance is sum of absolute differences between two vectors and can be written as:

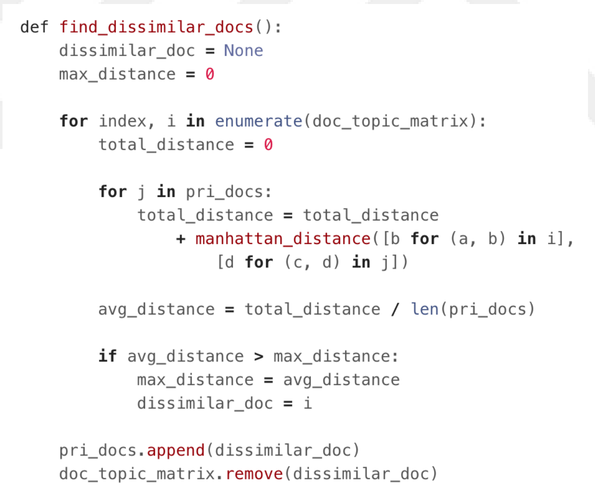
$$d_1(p,q) = \left\|p-q\right\|_1 = \sum_{i=1}^{n} |p_i - q_i| \tag{6.1}$$

where *(p, q)* are vectors.

In our case, vectors are topic membership probability values in the document-topic matrix for each document where documents are test cases. We calculate Manhattan distances between all test cases by iterating through document-topic matrix in Python. It is populated after LDA step.

6.3.3 Maximization Algorithm

We implement a greedy maximization algorithm. This algorithm finds the most dissimilar test case from entire test cases first and puts this test case to an initially empty list. Next, it finds the case that is most dissimilar to previously prioritized test cases and adds this test case to the same list. This algorithm continues until all test cases have been prioritized. Figure 6.4 represents our maximization algorithm written in Python.



```python
def find_dissimilar_docs():
    dissimilar_doc = None
    max_distance = 0

    for index, i in enumerate(doc_topic_matrix):
        total_distance = 0

        for j in pri_docs:
            total_distance = total_distance
                + manhattan_distance([b for (a, b) in i],
                    [d for (c, d) in j])

        avg_distance = total_distance / len(pri_docs)

        if avg_distance > max_distance:
            max_distance = avg_distance
            dissimilar_doc = i

    pri_docs.append(dissimilar_doc)
    doc_topic_matrix.remove(dissimilar_doc)
```

Figure 6.4 Maximization algorithm

pri_docs is a list for storing prioritized documents. It is empty initially. The most dissimilar document is added to *pri_docs* before running this algorithm. The most dissimilar document is found with similar approach to this algorithm. *doc_topic_matrix* stores document-topic probabilities. *manhattan_distance* is a simple function to calculate Manhattan distance between two vectors.

This is the last step of our TCP approach. Figure 6.5 summarizes the steps in our TCP implementation.

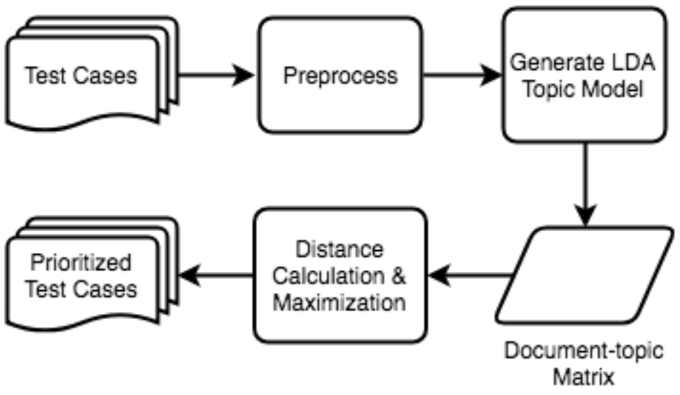
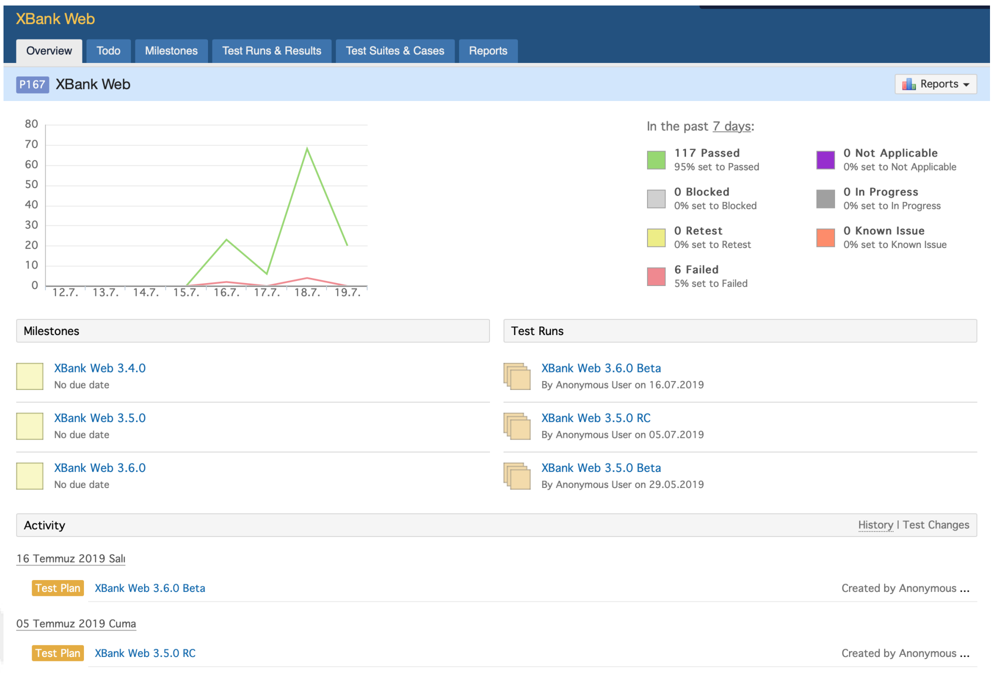


Figure 6.5 The architecture of our topic-based TCP approach

6.4 Experimental Results

6.4.1 Case Study

Test cases and their results are stored in TestRail test case management tool ("Test Case Management & Test Management Software Tool—TestRail," n.d.). To collect the faults, we looked at the test execution results in TestRail. Each test case has been exercised by testers, and the results have been reported in TestRail. Test engineers create different test runs in TestRail for each alpha, beta and release candidate testing. Figure 6.6 shows an example project dashboard including latest runs and activity. These test runs contain manually prioritized test cases in order. Recent three runs can be shown on "Test Runs" section. "Activity" section shows that the history of the test runs. There is also a chart that shows that the status of the test runs in the past 7 days.


XBank Web

Overview | Todo | Milestones | Test Runs & Results | Test Suites & Cases | Reports

P167 XBank Web — Reports

In the past 7 days:

🟩 117 Passed 95% set to Passed	🟪 0 Not Applicable 0% set to Not Applicable
⬜ 0 Blocked 0% set to Blocked	⬛ 0 In Progress 0% set to In Progress
🟨 0 Retest 0% set to Retest	🟧 0 Known Issue 0% set to Known Issue
🟥 6 Failed 5% set to Failed	

Milestones	Test Runs
XBank Web 3.4.0 No due date	XBank Web 3.6.0 Beta By Anonymous User on 16.07.2019
XBank Web 3.5.0 No due date	XBank Web 3.5.0 RC By Anonymous User on 05.07.2019
XBank Web 3.6.0 No due date	XBank Web 3.5.0 Beta By Anonymous User on 29.05.2019

Activity — History | Test Changes

16 Temmuz 2019 Salı

Test Plan XBank Web 3.6.0 Beta — Created by Anonymous ...

05 Temmuz 2019 Cuma

Test Plan XBank Web 3.5.0 RC — Created by Anonymous ...

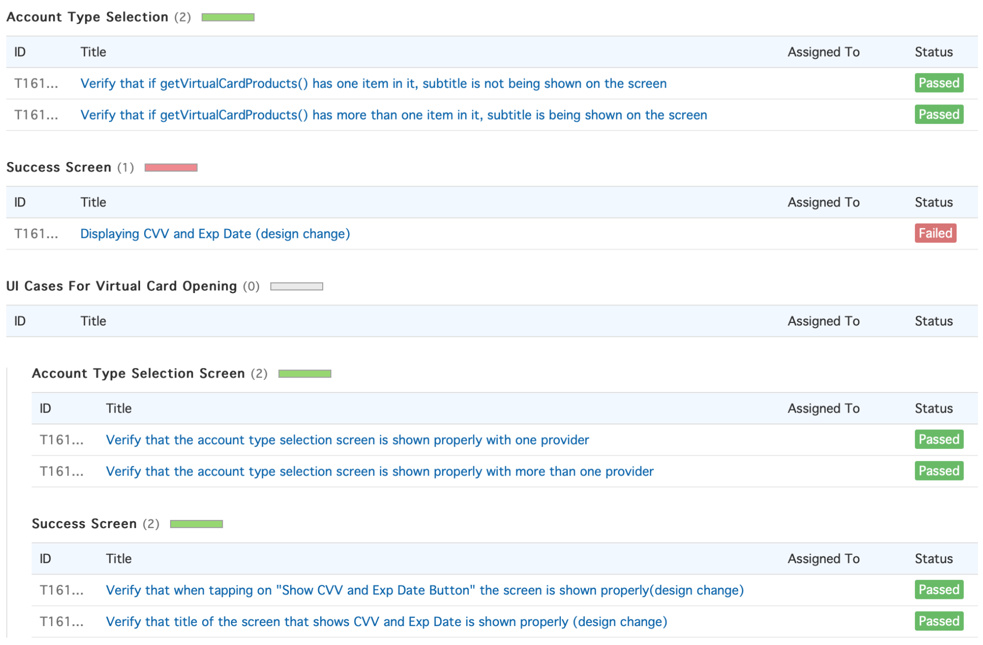
Figure 6.6 TestRail project dashboard

We collect 10 different tests run using TestRail API of our case study project. Table 6.7 shows the successful and failed test cases count of these runs. Total test case counts, and percent of faults are also presented in the table.

Table 6.7 Test case count results of test runs

Test Run	Successful Test Cases	Failed Test Cases	Total	Percent Faults
1	152	21	173	12.14
2	138	13	151	8.61
3	100	6	106	5.66
4	45	2	47	4.26
5	36	2	38	5.26
6	34	1	35	2.86
7	31	2	33	6.06
8	12	16	28	57.14
9	17	1	18	5.56
10	12	1	13	7.69

First of all, we fetch 10 test runs of the project and store them in the database. Then, fetch all test case scripts in these 10 runs ordered by manually prioritization. Also, we need to status (failed or successful) of each test case to evaluate the effectiveness of our TCP technique. Therefore, we fetch and store test case execution results in the database. Figure 6.7 shows an example TestRail interface including statuses of test cases in a test run. In this screen, test cases are grouped by features of application such as account type selection and virtual card opening. Test case ID, title and status can be shown in the list. Passed test cases are indicated as green and failed ones are indicated as red.



Account Type Selection (2)

ID	Title	Assigned To	Status
T161...	Verify that if getVirtualCardProducts() has one item in it, subtitle is not being shown on the screen		Passed
T161...	Verify that if getVirtualCardProducts() has more than one item in it, subtitle is being shown on the screen		Passed

Success Screen (1)

ID	Title	Assigned To	Status
T161...	Displaying CVV and Exp Date (design change)		Failed

UI Cases For Virtual Card Opening (0)

ID	Title	Assigned To	Status

Account Type Selection Screen (2)

ID	Title	Assigned To	Status
T161...	Verify that the account type selection screen is shown properly with one provider		Passed
T161...	Verify that the account type selection screen is shown properly with more than one provider		Passed

Success Screen (2)

ID	Title	Assigned To	Status
T161...	Verify that when tapping on "Show CVV and Exp Date Button" the screen is shown properly(design change)		Passed
T161...	Verify that title of the screen that shows CVV and Exp Date is shown properly (design change)		Passed

Figure 6.7 Statuses of test cases in a test run

Test engineers prepare test runs according to recent changes in the source code. Developers report impact analysis of the recent changes in the source to test engineers. Test engineers select related test cases with the help of impact analysis report. In this way, we try to verify that recent changes do not cause new faults. Test engineers sort these selected test cases by their priority at the same time. This manual sorting process does not perform well always as our experiments show. Sometimes, test cases are performed in random order due to the limited time. This is a much worse case than manual prioritization.

6.4.2 Evaluation

We need a metric to test the success of any TCP technique including the manual prioritization in our case. The average percentage of fault detection (APFD) metric is commonly used in the other studies in the literature.

Well-known APFD metric is used to test the effectiveness of TCP techniques, which was originally introduced by Rothermel et al. in (Rothermel, Untch, Chengyun Chu, & Harrold, 2001). APFD captures the average of the percentage of faults detected by a set of prioritized test cases. APFD is defined by following formula.

$$APFD = 100 * \left(1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}\right) \tag{6.2}$$

In this formula, *n* stands for the number of test cases, *m* stands for the number of faults and TF_i indicates the number of test cases, which must be executed before the fault *i* is detected. Larger APFD values indicate that increasing the effectiveness of TCP technique. In other words, more faults are detected with fewer test cases.

Note that, APFD calculation is only possible when prior knowledge of faults is available as shown in the formula. Therefore, APFD calculations are only used for evaluation.

TM is based on a machine learning algorithm and LDA uses a random seed to extract topic from a text collection. That's why LDA produces slightly different results in different runs. To solve this problem in our case study, we run our TCP method 30 times for every test run and take the mean of the APFD values. We present mean APFD values for topic-based TCP.

We need to automate all these steps in our experiment. We wrote Node.js scripts to achieve this. There are different scripts for each step including the data fetching from TestRail API. These Node.js scripts fetch data from TestRail as described in section 6.3.1 first. Then, the prioritization script calls our topic-modeling TCP Python script for each test run. Our last script is for APFD calculation. Topic-modeling TCP Python script returns prioritized list of test cases. APFD calculation script uses this list and manual prioritization list of test cases to calculate APFD.

We compared the proposed topic-based TCP technique in this thesis to manual prioritization technique using the APFD metric. APFD values of two different techniques are presented in Table 6.8.

Table 6.8 Mean APFD values of the experiment

	APFD		
Test Run	**Manual**	**Topic-based**	**Difference**
1	39.51	49.21	9.7
2	43.89	59.30	15.41
3	17.77	51.89	34.12
4	60.64	68.16	7.52
5	38.16	86.89	48.73
6	1.43	91.24	89.81
7	48.48	76.11	27.63
8	46.43	50.28	3.85
9	36.11	72.78	36.67
10	3.85	92.05	88.2

The results show that our proposed TCP method performs better than manual prioritization on average by a margin of APFD 36.16. The difference can be seen on line chart in Figure 6.8. We got higher mean APFD values on all test runs in our case study. This means if we prioritize test cases with this method before the execution, we can catch faults with executing fewer test cases. The results of the sixth and tenth test run is quite striking. They have one faulty test and that test is the last executed one in the manual prioritization. Our TCP method puts it to the first order in the prioritized list. This is the reason of major differences in the APFD values.

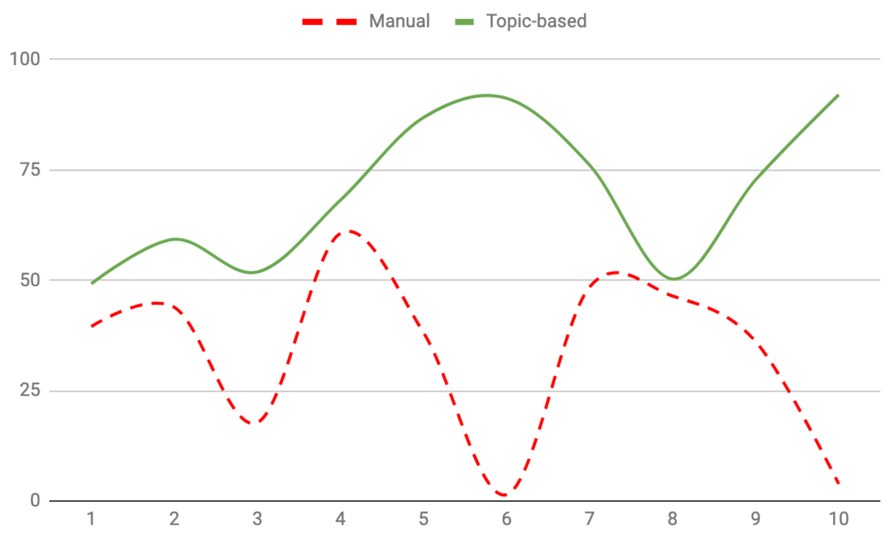


Figure 6.8 Line chart of APFD values for each test run

6.5 Advantages of Static TCP Techniques

Static TCP techniques have several advantages compared to execution-based and model-based TCP techniques.

Execution-based techniques need execution information (code coverage information) as described in Chapter 5. Similarly, specification models of SUT should be created for model-based techniques. Both processes require time and money. On the other hand, static TCP techniques do not require these data, this is the first advantage of static techniques.

Second, execution information does not need to be stored on the disk. This data may become too big on large systems. Since static techniques only use test cases, there is no need to store extra data.

Last advantage of black-box static techniques is that there is no need to update the execution information or specification models as the source code changes. Execution information and specification models should be updated when the requirements are changed. Black-box static techniques use test cases which are already being updated when the requirements are changed.

6.6 Future Works

In future work, we want to integrate information retrieval (IR) approaches to our TM approach. Currently, we don’t use any source code change information between versions when prioritizing test cases. We plan to use IR approaches to capture the relation between source code changes and test cases. Using this relation, we can find the related test cases for the new version of the software automatically, then prioritize them using our topic-based TCP method.

CHAPTER SEVEN
CONCLUSION

Delivering high quality software is an important goal for all software developers. To achieve their goal, too many test cases need to be executed on every new software release. Manual black-box testing is one of the most common software testing types, which has a high cost and takes too much time. Few TCP studies focus on making manual black-box software testing more efficient. In our work, we focus on black-box static prioritization in the context of manual testing.

We apply our approach to test case scripts of a commercial online banking project which is being developed with Agile methodologies and compare study results to manually prioritized test cases by test engineers. APFD values of our approach is higher than the manual prioritization approach. In other words, faults are detected earlier when our TCP technique is used for manual black-box testing.

The efficiency of the testing process is very critical because all possible faults should be detected with as few tests as possible within a limited testing budget. The experimental results show that the efficiency of the software testing processes is increased significantly by using the proposed TCP method.

REFERENCES

Blei, D. M. (2012). Probabilistic topic models. *Communications of the ACM*, *55*(4), 77.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2002). Latent dirichlet allocation. *Advances in Neural Information Processing Systems*, 601–608.

Boehm, B. W. (1978). *Characteristics of software quality*. Amsterdam: North-Holland Pub. Co.

Brooks, F. P. (1987). No Silver Bullet Essence and Accidents of Software Engineering. IEEE Computer, 20(4), 10–19.

Campbell-Kelly, M. (1995). Development and structure of the international software industry, 1950-1990. *Business and Economic History*, *73*(110), 38.

Cavano, J. P., & Mccall, J. A. (1978). A framework for the measurement of software quality. *Proceedings of the Software Quality and Assurance Workshop*, 133–140.

Griffiths, T. L., Steyvers, M., & Tenenbaum, J. B. (2007). Topics in semantic representation. *Psychological Review*, *114*(2), 211.

Hemmati, H., Arcuri, A., & Briand, L. (2013). Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, *22*(1), 1–42.

Hemmati, H., Fang, Z., Mäntylä, M. V., & Adams, B. (2017). Prioritizing manual test cases in rapid release environments. *Software Testing, Verification and Reliability*, *27*(6), e1609.

IEEE 610-1990—IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. (n.d.). Retrieved October 7, 2019, from https://standards.ieee.org/standard/610-1990.html

Jovanović, I. (2006). Software testing methods and techniques. *The IPSI BgD Transactions on Internet Research*, *30*.

Khan, M. E. (2010). *Different forms of software testing techniques for finding errors*. *7*(3), 6.

Khatibsyarbini, M., Isa, M. A., Jawawi, D. N. A., & Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, *93*, 74–93.

Korel, B., Koutsogiannakis, G., & Tahat, L. H. (2007). Model-based test prioritization heuristic methods and their evaluation. *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing - A-MOST '07*, 34–43.

Ledru, Y., Petrenko, A., Boroday, S., & Mandran, N. (2012). Prioritizing test cases with string distances. *Automated Software Engineering*, *19*(1), 65–95.

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality*. Retrieved from GENERAL ELECTRIC CO SUNNYVALE CA website: https://apps.dtic.mil/docs/citations/ADA049014

Naik, K., & Tripathy, P. (2011). *Software testing and quality assurance: Theory and practice*. New York: John Wiley & Sons.

Natural Language Toolkit—NLTK 3.4.5 documentation. (n.d.). Retrieved October 7, 2019, from https://www.nltk.org/

Naur, P. (1968). *Software Engineering-Report on a Conference Sponsored by the NATO Science Committee Garimisch, Germany* (p. 136). Retrieved from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

Pfleeger, S. L., & Kitchenham, B. (1996). Software Quality. *IEEE Software*, *12*(21), 20.

Porter, M. F. (2006). An algorithm for suffix stripping. *Program*.

Řehůřek, R., & Sojka, P. (2011). Gensim—Statistical semantics in python. *Statistical Semantics; Gensim; Python; LDA; SVD*.

Rothermel, G., Untch, R. H., Chengyun Chu, & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, *27*(10), 929–948.

Sampath, S., Bryce, R. C., Viswanath, G., Kandimalla, V., & Koru, A. G. (2008). Prioritizing User-Session-Based Test Cases for Web Applications Testing. *2008 International Conference on Software Testing, Verification, and Validation*, 141–150.

Sawant, A. A., Bari, P. H., & Chawan, P. M. (2012). Software Testing Techniques and Strategies. *International Journal of Engineering Research and Applications*, *2*(3), 7.

Shewhart, W. A. (1930). Economic Quality Control of Manufactured Product. *Bell System Technical Journal*, *9*(2), 364–389.

Silva Simao, A., De Mello, R., & Senger, L. (2006). A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network Architecture. *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, 93–96.

Snowball: A language for stemming algorithms. (n.d.). Retrieved October 6, 2019, from http://snowball.tartarus.org/texts/introduction.html

Test Case Management & Test Management Software Tool—TestRail. (n.d.). Retrieved February 28, 2019, from https://www.gurock.com/testrail

Thomas, S. W., Hemmati, H., Hassan, A. E., & Blostein, D. (2014). Static test case prioritization using topic models. *Empirical Software Engineering*, *19*(1), 182–212.

van Vliet, H. (2008). *Software Engineering: Principles and Practice* (Vol. 13). New York: John Wiley & Sons.

Wong, W. E., Horgan, J. R., London, S., & Agrawal, H. (1997). A study of effective regression testing in practice. *Proceedings The Eighth International Symposium on Software Reliability Engineering*, 264–274.

Yoo, S., Harman, M., Tonella, P., & Susi, A. (2009). Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis - ISSTA '09*, 201-212.

Zhang, L., Zhou, J., Hao, D., Zhang, L., & Mei, H. (2009). Prioritizing JUnit test cases in absence of coverage information. *2009 IEEE International Conference on Software Maintenance*, 19–28.