2020

VALUE ESTIMATION OF SOFTWARE FUNCTIONAL TEST CASES

Yao Shi

VALUE ESTIMATION OF SOFTWARE FUNCTIONAL TEST CASES

by

Yao Shi

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Major: Business Information and Technology

The University of Memphis

May 2020

Acknowledgments

Pursuing this Ph.D. has been a truly life-changing experience for me. It would not have been possible without the support and guidance that I received from many people.

I am indebted to my parents and would like to say a heartfelt thank-you to my Mom and Dad for always believing in me, for the unconditional love and sacrifice, and for the unending encouragement and support that help me in chasing my dreams throughout the last decade.

I would like to express my deepest and sincere gratitude to my research advisor and dissertation committee chair, Dr. Mark Gillenson, for giving me the opportunities to do research with him and providing invaluable personal and professional guidance. Numerous tableside discussions and chats with him in his office where I gained the knowledge and wisdom were unforgettable. His vision, sincerity, trust, and patience have deeply inspired me. It was a great privilege and honor to work and study under his mentoring.

I cannot express enough gratitude and thanks to my committee members, Dr. Robin Poston and Dr. Euntae "Ted" Lee, for their unwavering support throughout my dissertation. Their encouragement when my study and research got rough is much appreciated and duly noted.

My dissertation could not have been accomplished without constant feedback of my committee member, Dr. Xihui "Paul" Zhang. I offer my sincere appreciation for his friendship and patience in training me during the entire period of pursuing my doctoral

degree. The countless times he taught me in improving all the details of methodology to carry out and to present the research will not be forgotten.

My special praise and respect go out to Dr. Thomas Stafford for his professional advice, inspiration, trust, and empathy. He has taught and supported me more than I could ever give him credit. He has shown me, by example, what a good research scholar should be.

I am thankful to all faculty, staff, and fellow Ph.D. students in the Department of Business Information and Technology for their collaboration and support in different ways. It truly has been a very good time studying in the department.

Last, I would like to cite one of my favorite songs, "You Raise Me Up," to thank all my family members, professors, and friends who have supported me on this precious academic journey.

"**You** raise me up so I can stand on mountains;

You raise me up to walk on stormy seas;

I am strong when I am on **Your** shoulders;

You raise me up to more than I can be!"

Abstract

Software testing is becoming more and more critical to ensure that software will function properly in the production environment. Consequently, the effort, time, and funds invested in software testing activities have been increased significantly. However, these resources cannot meet the increasing demand of software testing. As such, managers have to allocate testing resources to the test cases that are more critical to uncover defects. This study builds a value function that can quantify the value of a test case and thus provide an approach in selecting key functional test cases. Following the guidance of case study research and using an innovative methodology to develop a mathematical function, we took three steps to develop a value function of software functional test cases. First, we built an initial value function based on a systematic analysis of the pertaining literature and theoretical background. Next, we interviewed industrial professionals and managerial staff who are working in testing to provide expert comments and give practical feedback on the initial value function. Finally, based on an in-depth analysis of the comments and feedback from the interviews, we revised and finalized the value function by incorporating some of the new factors that emerged from the interviews and modifying some of the initial factors that varied in meaning according to the viewpoints of the interviewees. This finalized value function can play a significant role in prioritizing test cases and addressing the resource constraint issues in software testing.

Keywords—value estimation; test case; software testing; resource constraint; case study.

CONTENTS

List of Tables

Table Page

List of Figures

Figure Page

LIST OF ABBREVIATIONS

AI Artificial Intelligence

AU Amount of Use

CC Code Coverage

CEO Cost to Determine Expected Outcome of a Test Case

CER Cost to Evaluate Test Results

CFC Cost to Determine Failure Category

CI Complexity Issues

CIV Cost to Create a Test Case Input Values

CMD Cost to Manage a Defect

CME Cost to Collect and Record Test Metrics

COCOMO Constructive Cost Model

CRF Cost to Resolve Test Case Failure

CRR Cost to Record and Report Test Results

CRT Cost to Run a Test Case

CTC Cost of a Test Case

DI Dependency Issues

ER External Risk

EV Execution Valve

FC Function Coverage

FIPS Federal Information Processing Systems

IEEE Institute of Electrical and Electronics Engineers

IR Internal Risk

IT Information Technology

MOS Microsoft Office Suite

PI Personnel Issues

PR Production Risk

PTI Previous Testing Issues

RI Requirements Issues

ROI Return on Investment

SUT Software under Test

TEI Test Environment Issues

TF Test Frequency

TI Technology Issues

TR Technical Risk

TU Test Utility

VI Vender Issue

VTC Value of a Test Case

XP Extreme Programming

CHAPTER 1

INTRODUCTION

1.1 Software Testing Practical Issues

As software applications permeate everywhere in the world, people are becoming more sensitive to the validity and reliability of software applications (Juristo et al., 2006). Defects in the applications may result in tremendous monetary loss, time lost, and even innocent death (Felderer & Ramler, 2014). According to a recent annual report by Tricentis, a leading company providing software testing solutions, about 606 major software failures from 314 companies occurred around world in 2017. These failures

caused about $1.7 trillion financial losses, 268 years of the cumulative downtime, and affected 3.6 billion people (Tricentis, n.d.). Billions of dollars are invested in software development every year around the world (Cresswell, 2004), and approximately 50 percent of the total elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed in a typical software development project (Boehm & Papaccio, 1988; Myers et al., 2011). Despite that software testing has been considered as an important step in the software development life cycle to assure software quality, the defects still cannot be entirely eradicated due to inadequate testing (Tricentis, n.d.; Whittaker, 2000).

In order to reveal the causes of inadequate software testing, we identified five practical issues in software testing based on Whittaker's (2000) study. Since a series of testing for different purposes (e.g., functional testing, performance testing, and security testing) needs to be carried out before a software program is released (Mathur, 2013), we only focus on functional testing in this study because functional testing is to examine the functionality of a program and is also the fundamental testing for the other types of testing such as integration testing and system testing.

1.1.1 Untested Code

An application usually cannot be released until the appropriate testing has been conducted. However, as software becomes much larger and more complex, some code may not be tested or may be untestable before the application is released due to time constraints or testing techniques not being able to keep pace with the software development techniques (Whittaker, 2000). To avoid late delivery, a very common strategy in practice is to test the critical code with important functions and features while

delaying the testing of unimportant or untestable code. Despite this method benefiting the pace of software development, software quality is put at risk as the amount of untested code increases (Felderer & Ramler, 2014). Therefore, test engineers always face tough decisions in striking an acceptable balance between the pace of software delivery and software quality.

1.1.2 Untested Combination of Input Values

Multiple input variable values are typically needed when testing a program. As the number of variables and values of individual variables increase, the combinations of input values become more complex in the set of test cases. In this situation, using a large number of test cases is neither feasible nor valuable (Goodenough & Gerhart, 1975; Myers et al., 2011). Similar to the aforementioned method, software testers adopt the same strategy to conduct testing where the main or critical combinations of the input values are the focus. Although this maximizes the yield from the testing process, some uncommon combinations of the input values that may lead to software crashes might never be tested.

1.1.3 Untested Path

The source code of a program generates multiple executable paths. In practice, users sometimes follow different sequences which may not be fully considered in the software design. Such scenario often occurs, especially when the users are not familiar with the specific operation of the program, whereas test engineers usually conduct critical path tests to try to pinpoint important software faults (Hass, 2014). However, software programs typically consist of multiple functions and features involving numerous paths. Trying to test all of the paths is not feasible. In order to reduce redundant tests and increase test efficiency, software engineers usually conduct only critical path tests which

check the paths that are most likely to be used rather than all the paths in a program (Jorgensen, 2018). As a result, this test strategy may create a potential risk where the program could crash due to a defective path triggered by users if the path has not been tested.

1.1.4 Untested Operating Environment

Nowadays, a popular application is usually made in several versions which are compatible in different platforms and operating environments. For instance, Microsoft Office Suite (MOS) is one of the most popular productivity applications around the world. As it has evolved in the last thirty years, MOS has been developed in many versions that can be used on PC, Mac, and mobile devices. Although it is a highly mature application, MOS is still continuously being improved in increasing the compatibility on different platforms and operating environments ("History of Microsoft Office," 2020). Moreover, the users' operating environments are much more complicated and dynamic than they used to be. Software engineers find it almost impossible to simulate all possible conditions to test a software program. In other words, users' operating system configurations are so diverse that no one has a way of capturing all the configurations for testing (Jorgensen, 2018). For instance, users might install different peripheral devices in their systems; or the operating environment might be changed as different tasks are executed at the same time, even though the operating system configurations are the same. Therefore, test engineers only simulate primary operating environments which means that their testing strategy might lead to software faults occurring in the untested operating environments (Juristo et al., 2006).

1.1.5 Defective Testing Procedure

Differing from prior issues which are constrained by non-technical factors/resources such as time and computing capability, the defective testing procedure issue is constrained by technical factors. In fact, a software program becomes more complex as its size increases and a large number of features is required to be integrated within one system (Felderer & Ramler, 2014). In the meantime, software testing also faces big challenges because those testing procedures derived from traditional development methods are not able to detect software faults efficiently or effectively if the program is developed through new methods or new languages (Burnstein, 2006). Under this situation, the testing procedures need to be either updated or replaced by new procedures to prevent hidden bugs in a program.

1.1.6 Summary

Inadequate software testing usually results in defective applications and negative outcomes. Inspired by Whittaker's (2000) study, we identified five primary software testing practical issues (untested code, untested combinations of input values, untested paths, untested operating environments, and defective testing procedures) causing the inadequate software testing. And these five testing issues are rooted in resource constraints and technical constraints in software testing. On the one hand, a company might not have adequate resources such as budget, time, or personnel to run sufficient tests, resulting in four testing issues (untested code, untested combinations of input values, untested paths, and untested operating environments). On the other hand, a company might have enough resources but without key technical support such as sophisticated algorithm, powerful testing tools, or expert testing engineers, incurring the issue of defective testing procedures. Considering most practical software testing issues result from resource

constraints (see Figure 1), we therefore focus on exploring the value of software test cases in this study, which we think is an effective method to deal with resource constraints of software testing.

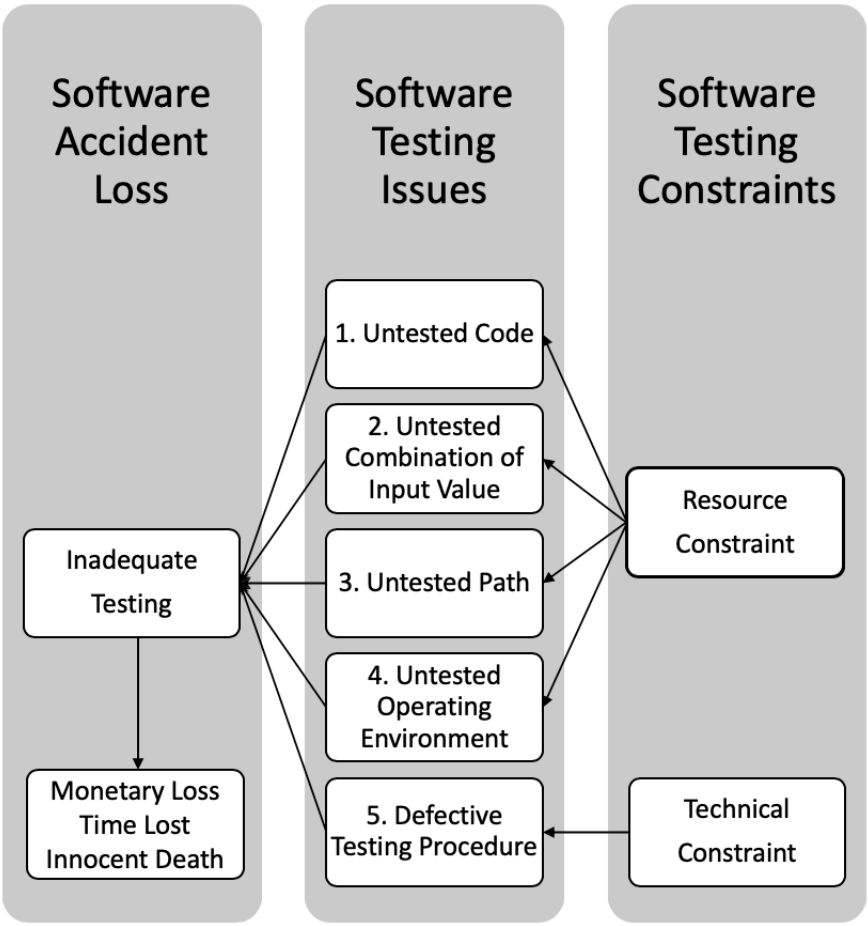


Figure 1. Practical Issues in Software Testing

1.2 Software Testing Research Issues

There are fundamental disagreements regarding the resource allocation in software development. Although a huge amount of dollars is spent in software development and testing, resource allocation is still considered as the dominate cause of software development failures such as defected software and aborted software development

projects (Tassey, 2002). Researchers ascribe the software failure to insufficient requirement and design analysis in the early stage or the invalid software testing regarding the requirements elicitation (Boehm, 1981; Charette, 2005; Jalote & Vishal, 2003); in practice, however, most of the resources are allocated to coding due to release pressure (Yiftachel et al., 2011). Moreover, there is a widely accepted view of software testing: exhaustive software testing is impossible due to resource constraints such as limited time, funds, and personnel (Myers et al., 2011). To deal with the resource constraint issue in software testing, two main research streams emerged during the last decades.

1.2.1 Effort Estimation

The first stream, effort estimation, mostly emerged in the 1980's and 1990's (Molokken & Jorgensen, 2003). Originally, researchers attempted to estimate the effort consumed in the whole process of software development where software testing is a part of the entire development process. The main purpose of the effort estimation is to learn the extent of the deviation between an actual software development project and its original plan, regarding cost, schedule, and functionality (Jorgensen, 2004). In this time period, most effort estimation studies were developed for the entire software project, while the effort estimation of software testing was considered as a part of the effort.

Table 1 shows the methods of software estimation that have been studied in academia and used in practice. Expert judgement, which relies on intuition, experience, historical data, and process guideline (Jorgensen, 2004, 2005), is the most frequently used method. The strength of this method is that it can be applied in almost any context without a high threshold because it heavily relies on the expert's experience. Additionally, it can be applied in software size, effort, schedule, and cost estimation. The weakness of this

method is that the accuracy might be very low especially when the software is too complex. To increase the estimate accuracy, experts usually use historical data of the similar software projects to assist the estimation. To estimate software project size, source lines of code and function points are the two common methods. Source lines of code method measures the size by counting the number of lines in the program's source code (Albrecht & Gaffney, 1983). Since line of code is a physical entity, this method is feasible and reliable. However, same function in a program could be written differently, it cannot count the size while considering created functions. Function points is the method that expresses the amount of business functionality a program provides (Dreger, 1989). This method avoids the issue that using large number of lines of code to create relative fewer functions in a program. The model-based method is another main approach. A well-known such model is the Constructive Cost Model (COCOMO) developed by Boehm (1981). To accurately estimate the cost of software, the model's parameters are derived from historical projects and rely on size estimation through source lines of code. The method estimates not only the project schedule but also cost of the software project. Although this method enables to provide a relatively accurate estimation, it still cannot provide the adequate information about the critical or valuable process deserving to be invested more resources.

Table 1. Effort Estimation Approaches

Estimation Approach	Description	Estimation Type	Major Study
Expert judgement	Relies on expert intuition and experience.	Project size, schedule, cost	Jorgensen, 2004, 2005
Source lines of code	Counting the number of lines in the program's source code.	Project size	Albrecht & Gaffney, 1983

Function points	Express the amount of business functionality a program provides to users.	Project size	Dreger, 1989
Model-based (e.g., COCOMO)	A model based on source lines of code to estimate software project schedule and cost.	Project schedule, cost	Boehm, 1981

1.2.2 Value-Based Estimation

Despite the fact that the effort estimation methods provide some information regarding expenditures of software development including software testing, it is still difficult for software engineers to judge what software testing should be performed. A large effort in software testing may or may not increase software quality. Therefore, value-based estimation is intended to apply the effort expended in the most effective way (Boehm, 2006). Along with the value-based view, Biffl et al. (2006) maintain that the major value arises from a few software testing processes. Software engineers thus take into account various factors in selecting software test as well as corresponding test cases to attempt to achieve a maximal contribution. However, assembling an optimal portfolio of software tests drawn from an extensive list of available testing approaches is not an easy task. Software testing as a support activity intertwined with other parts of the software life cycle cannot deliver a significant contribution to the software development process unless the particular high-value software testing activities are identified and implemented (Hass, 2014).

Several dimensions have been studied in generating value-based software (see Table 2). First, value-based requirements. Since the objectives of systems as well as following steps rely on the requirements, identifying a system's critical stakeholders,

eliciting their valuable requirements, and reconciling the requirements to the system are critical. To this end, Wohlin and Aurum (2006) use survey to identify the critical stakeholders as well as requirement criteria. Second, value-based architecting. This dimension focuses on reconciling the system objectives with achievable architectures. Kazman et al. (2001) built an economic model of architectural decision making, which is based on cost benefit analysis of system quality attributes. Third, value-based design and development, which involves inheriting the system objectives and value considerations into system design and development. Van Solingen (2004) used return on investment (ROI) rather than other complicated models to measure the improvement in the software development process. Last, value-based verification and validation process, which involves testing, is considered as an investment activity. It focuses on ensuring the verification and validation process to satisfy value objectives. Felderer and Ramler (2014) argue that risk should be considered when planning software testing. Although risk is not easily to measure in practice, neglecting the risk from software testing would decrease the effectiveness of software testing since the resource is limited. Therefore, they propose a process model to integrate risk analysis and software testing.

Table 2. Value-Based Estimation Approaches

Estimation Approach	Description	Estimation Type	Major Study
Survey	Identifying a system's critical stakeholders, eliciting their valuable requirements, and reconciling the requirements to the system.	Value-based requirements	Wohlin & Aurum, 2006
Cost benefit analysis	Reconciling the system objectives with achievable architectural.	Value-based architecting	Kazman et al., 2001

Return on investment analysis	Inheriting the system objectives and value considerations into system design and development.	Value-based design and development	Van Solingen, 2004
Risk-based testing	Ensuring verification and validation process satisfies value objectives; considering verification and validation process as investing activity.	Value-based verification and validation	Felderer & Ramler, 2014

1.2.3 Summary

To deal with the resource constraint issue in software testing, effort estimation and value-based estimation are two research streams that are formed for allocating testing resources. However, they have some critical drawbacks: (1) effort estimation only focuses on the resources expended in the process of software development which includes software testing, and it cannot assist software engineers in choosing software testing processes which contribute the most to software quality; and (2) existing studies of value-based estimation focuses on value of each process of software development rather than software testing. Therefore, those value-based estimation studies cannot provide a break-down or specific estimation within software testing.

Given the shortcomings, we cannot simply use existing studies from the prior two streams for solving the resource constraint issue in software testing. Therefore, it is necessary to establish a new mechanism in value-based estimation focusing on software test cases. Because generating test cases is a critical step in functional software testing no matter the program is developed in the traditional waterfall paradigm or the agile paradigm. This new method would make significant contributions to selecting test cases and systematically allocating resources in functional software testing.

1.3 Research Objective and Research Question

As the primary practical issues in software testing are identified, we find resource constraint is the root cause of inadequate testing. Given that exhaustive software testing is impossible (Myers et al., 2011), how to maximize the efficiency and the effectiveness of software testing with limited resources becomes the most important question in the software testing domain (Juristo et al., 2006). Although researchers have dedicated to solving the problem of resource constraint in software testing from different perspectives in the last few decades (Biffl et al., 2006; Boehm, 2006; Felderer & Ramler, 2014; Wohlin & Aurum, 2006), the shortcomings of the prior studies from the two research streams indicate that the existing approaches cannot appropriately address the problem. This is because that the software testing methods either lag behind software development methods or just take into account software engineering factors which cannot provide adequate guidance for improving software testing (Juristo et al., 2006; Talby et al., 2006).

Therefore, the research objective of this study is *to explore a new mechanism involving the comparative value of test cases to increase the efficiency of software testing*. Since test cases are the core part of software testing and are also the critical steps to optimize the efficiency of the software testing, all else being equal, choosing the test cases producing relative high value can optimize the software testing in a resource constrained environment (Biffl et al., 2006). To specify the research objective, we intend to develop a function that assigns a value of a test case for the purpose of comparing it with the value of other test cases. We therefore initiate the research question surrounding the evaluation of software test cases: *What is the relative value of a functional test case in software testing?*

The potential contribution of this study is threefold. First, the function we develop reveals the essence of value for a context-specific test case. In this way, individual assessment for optimization decision making can be enhanced. Second, the value function of test cases fills a notable gap in the literature, as there currently exists no specific method to systematically determine and justify the value of test cases. Knowing this will enhance the capabilities of software testing managers. Last, the specific exploration of the notion of test cases establishes an important reference point for software testing as well as systems development in the critical corporate governance task of resource allocation.

1.4 Research Structure

This study is organized into nine chapters. Chapter 1 introduces the current practical issues and research issues of software testing. Following that, research objectives, research questions, and the structure of this study are presented. Chapter 2 explores the research foundation from nature of software testing and test cases, nature of value, nature of cost in software test cases. Chapter 3 describes the research process, methodology, data collection, and coding process. Chapter 4 delivers the initial value function of test cases based on a systematic analysis of the pertaining literature and theoretical foundation. The function presents the value of the test cases from two dimensions consisting of four levels: business dimension (risk level and cost level) and software engineering dimension (application level and unit level). Chapter 5 demonstrates the interview results, which are the comments upon the given initial value function that we collected from industrial testing professionals. Each factor in the function is then analyzed and the new factors derived from the interviews are illustrated as well. In Chapter 6, we

deliver the final form of the value function of test cases. The components of each factor in the final value function, factor scoring, factor weighting, and calculating mechanism of the final value function are introduced. Chapter 7 compares the final value function of test cases with other value determination rubrics applied in software testing. In Chapter 8, we provide guidance for the general application for the final value function. Chapter 9 addresses the limitations of the final value function and offers future research directions in improving the function.

CHAPTER 2

RESEARCH FOUNDATION

To address the research question, we aim at building a function to calculate the value of software test cases. Yet a sound function cannot be established without a solid foundation for the concepts. In other words, the essences of software testing, test cases, and value are still vague because they have been discussed for different purposes and in different contexts (Biffl et al., 2006; Boehm, 2006; Gelperin & Hetzel, 1988; Jorgensen, 2018; Mathur, 2013; Myers et al., 2011; Perry, 2007). Therefore, adopting those concepts into the field of software testing without adapting them would result in failure in distinguishing the value of test cases. Therefore, before establishing the function, we attempt to build the research foundation regarding the core concepts, including the nature of software testing and test case, nature of value, and nature of cost in software test cases.

2.1 Nature of Software Testing and Test Case

2.1.1 Nature of Software Testing

Since the earliest article on program checkout was written by Alan Turing in 1949 (Gelperin & Hetzel, 1988), software testing has been growing for almost eight decades and the nature of software testing has been changing as the use of digital computers increased and diversified. Based on Gelperin and Hetzel's (1988) narration of software testing growth, we added a new growth stage of software testing starting from 1990's (see Table 3).

Table 3. Growth Stages of Software Testing

Period	Category	Interpretation of Software Testing
- 1950's	The Debugging-Oriented Period	Testing and debugging are used interchangeablely. Selecting test cases relies on programmers' experience.
1950's – 1970's	The Demonstration-Oriented Period	Testing focuses on "make sure the program solves the problems." Debugging focuses on "make sure the program runs."
1970's – 1980's	The Destruction-Oriented Period	Testing is concerned with revealing the faults existed in the program. Debugging is concerned with locating and fixing those faults.
1980's -1990's	The Evaluation & Prevention Oriented Period	Software testing is integrated into the evaluation phase for assessing how well the products in each phase of software life-cycle meet their requirements.
1990's -	The Test-Driven-Oriented Period	Testing lead and intensively interact software development. Testing activities widely spread among development (sprint) and closure phases.

Adapted from Gelperin & Hetzel (1988)

In the debugging-oriented period (prior to 1950's), testing focused on hardware and programs were written and checked out by the programmers until all the outstanding

bugs had been identified and fixed (Gelperin & Hetzel, 1988; Turing, 1950). There is no clear distinction between testing and debugging, resulting in these two terms being used interchangeably. The criteria used for selecting test cases are entirely ad hoc and relied exclusively on programmers' experience and understanding of the system.

In the demonstration-oriented period (1950's to 1970's), testing and debugging were considered as different activities. Testing focuses on "make sure the program solves the problems" and debugging focuses on "make sure the program runs" (Baker, 1957). In other words, testing ensures that the program conforms to its requirements whereas debugging attempts to prevent the program from any crashes.

In the destruction-oriented period (1970's to 1980's), the description of testing in Myers et al.'s (2011) book has gained wide acceptance where testing is defined as "the process of executing a program with the intent of finding errors." With that, testing and debugging were differentiated and demonstrated in new meanings. Testing is concerned with revealing the faults existed in the program, but debugging is concerned with locating and fixing those faults (Deutsch, 1981; Miller & Howden, 1981).

In the evaluation & prevention-oriented period (1980's to 1990's), several standards were proposed to pave the way of regulating the testing activities. The U.S. National Bureau of Standards issued a guideline in 1982 (Neumann, 1982), which specifically targeted at federal information processing systems (FIPS). Software testing is integrated into the evaluation phase for assessing how well the products in each phase of software life-cycle meet their requirements. Following that milestone, the Institute of Electrical and Electronics Engineers (IEEE), the world's largest technical professional organization, published the "IEEE standard for software verification and validation plans"

in 1986 (IEEE, 1986). This standard provides uniform and minimum requirements for the format and content of software testing in evaluating each phase of the software project. The goal of those standards is to identify and correct the faults in the software in an early stage.

In the test-driven-oriented period (after 1990's), testing has been shifting from evaluating and preventing function to leading or intensively interacting software development. In 1997, Ken Schwaber published SCRUM (Schwaber, 1997), an entirely new software development methodology differing from the traditional waterfall method. SCRUM assumes that the systems development process is an unpredictable, complicated process rather than a well understood process that can be perfectly planned, estimated, and successfully completed. Testing activities widely spread among development (sprint) and closure phases. In the sprint phase, all the development activities are assessed continuously by testing and adequate controls and responses put in place. Extreme Programming (XP), another software development methodology with similar philosophy of SCRUM, was released by Kent Beck in 1999 (Beck, 1999). Rather than planning, analyzing, designing, implementing, and testing in conventional software development process, XP blends all these activities in several iterations and then breaks the iterations down into tasks which are estimable and testable. To implement a task, two programmers are paired and write their own tests before they start coding. This reverse process not only shortens the feedback time to the programmers but also provides a dynamic way for software development.

2.1.2 Nature of Software Test Case

Although software testing is illustrated in prior sections, unscrambling the nature of test cases is necessary because software testing and test cases are a cohesive unit. The deeper narrowing test cases, the better understanding software testing and estimating the value of test cases. In this section, we addressed the definition and category of a test case.

In the international standard of systems and software engineering (ISO/IEC/IEEE, 2017), a test case is defined as *"a set of test inputs, execution conditions, and expected results developed for a particular objective*; and software testing is demonstrated as *an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component."*

Obviously, as the two definitions explicitly described, test cases are developed for different software testing and inherently serve for the corresponding software testing on a particular objective. Therefore, an appropriate method to categorize test cases is to classify the test cases based on existing software testing classification. After reviewing the primary classification of software testing (e.g., Jorgensen, 2018; Mathur, 2013; Myers et al., 2011; Perry, 2007), we found that there was no consensus on this: some types of testing overlap, while others are referred to in different terms. To build a systematic classification of typical test cases, we chose two primary classifiers for the test case classification: (1) source of test generation, and (2) lifecycle phase.

In terms of source of test generation, functional test cases are classified into the test cases generated from black-box testing and those generated from white-box testing. These two types of testing are designed to ensure that the system requirements and specifications are achieved (Perry, 2007). Therefore, the purpose of the test cases

generated from black-box testing and white-box testing is to test whether the program functions work correctly. The typical black-box test cases incorporate equivalence partitioning test cases, boundary-value analysis test cases, cause-effect graphing test cases, and error guessing test cases. White-box test is also known as structural testing, which is concerned with the degree to which test cases exercise or cover the logic of the program (Mathur, 2013; Myers et al., 2011). The typical white-box test cases incorporate statement coverage test cases, decision coverage test cases, condition coverage test cases, decision-condition coverage test cases, and multiple-condition coverage test cases (see Table 4).

Table 4. Test Case Category (Source of Test Generation)

Category	Type of Test Case
Black-box	Equivalence partitioning test case
	Boundary-value analysis test case
	Cause-effect graphing test case
	Error guessing test case
White-box	Statement coverage test case
	Decision coverage test case
	Condition coverage test case
	Decision-condition coverage test case
	Multiple-condition coverage test case

In terms of lifecycle phase, test cases are classified into five types to test the corresponding phases (Mathur, 2013). In coding phase, unit testing cases are usually applied to test the individual units or components of a software. In integration phase, integration testing cases take place to test several individual modules which are combined together. Integration testing cases are usually created after unit testing. In system integration phase, system testing cases are to test a complete and fully integrated software

product. In the maintenance phase, most regression testing cases are derived from previous test cases and executed automatically because regression test retests the existing software applications to make sure that a change or addition has not broken any existing functionality. In the last post system/pre-release phase, beta-test cases take place to test the software prior to commercial or official release. The category and typical example of test cases are listed in Table 5.

Table 5. Test Case Category (Life Cycle Phase)

Category	Type of Test Case
Coding	Unit test case
Integration	Integration test case
System integration	System test case
Maintenance	Regression test case
Post system/pre-release	Beta-test case

In general, functional testing involves all the life cycle phases of software development from coding phase to pre-release phase, but performance testing and security testing usually play roles in the middle or end stage of software testing, such as module integration and system integration. We focus on functional test cases (see an example of a functional test case in Appendix A) in this study and attempt to build a value function which is able to assess the functional test cases in all the life cycle phases.

2.2 Nature of Value

To uncover the value of software test cases, another critical step is to probe and define the nature of value. We therefore cascade down the nature of value from its origin

in philosophy to the extended regions in business, information technology, software engineering, software testing, and test cases.

2.2.1 Value in Philosophy

Value in philosophy presents an original meaning. The Cambridge dictionary defines "value" as "the amount of money that can be received for something; or the importance or worth of something for someone" (Cambridge Dictionary, n.d.). In axiology, philosophical inquiry into value is structured around three related concerns. "First, determining what we are doing when we ascribe value to the entities. Second, saying whether value is subjective or objective. Last, specifying what things are valuable or good" (New World Encyclopedia, 2016).

From the above definitions, we conclude that the meaning of value consists of three dimensions that need to be taken into account when exploring the nature of value in the following perspectives: business, information technology, software engineering, software testing, and software test cases (Cambridge Dictionary, n.d.; New World Encyclopedia, 2016). First, value refers to the benefit which is generated from certain activities. The benefit could be measured as tangible things such as money and also as intangible things such as prevented risks. Second, value is only related to its stakeholders. In other words, the benefit created from certain activities is only valuable to relevant entities rather than all the entities. Last, value needs to consider both benefit and the corresponding cost.

2.2.2 Value in Business

Value in business has been portrayed by Harvard's Michael Porter, who is well-known in the business domain for his notions of value analysis. He mainly demonstrates

the implication of value from the firm level, noting that it is the amount buyers are willing to pay for what a firm provides (Porter, 1985). Generic competitive strategies, then, revolve around creating value for buyers who are willing to pay more than the cost of providing that value. Although one of the basic strategies is, indeed, cost-based, Porter is of the opinion that value rather that cost is the best factor to use in analyzing competitive position (Porter, 1985; Porter & Millar, 1985).

In a broad term, value is created through products or services which are transactable and acceptable by customers (Porter, 1985; Porter & Millar, 1985). The purpose of a business is to create value through producing products or providing services. In general, business value consists of two dimensions (Porter & Millar, 1985). The first dimension is the business value in firm level. Porter's value chain breaks down the production process in a firm into several connecting activities. The primary activities involve inbound logistics, operations, outbound logistics, marketing and sales, and service. In inbound logistics activities, the materials are received, stored, and distributed to designated places. All the raw materials, labor, as well as other necessary things are converted into products or services in operations activities. In outbound logistics activities, the final products are moved from the end of the production line to the end users. In marketing and sales activities, selling products or services, communicating with customers, and researching on competitors are the main purposes. To keep all the products or services working effectively after being sold is the primary activities in service stage. Other than the primary activities that enable to add the value in the production, the support activities play a complementary role in facilitating the primary activities to add value in the business. The support activities include firm infrastructure, human resource

management, technology, and procurement. No matter what activities to be taken in the firm, every activity is supposed to add value (Porter & Millar, 1985).

The second dimension is the business value in industry level, which is known as value system. In value system, value is not only created within a firm but also added and delivered among the different entities from upstream to downstream. In upstream, suppliers provide valuable raw materials or components to production companies (Porter & Millar, 1985). The production companies' products usually pass through downstream channels on their way to the ultimate buyers.

To conclude, value in business is realized through providing products or services to fulfill the customers' requirements and obtain the return for the business. During the business process, value is added from the very beginning step to the very end step. In order to optimize the value creation, the value and cost occurred in each step should be evaluated that can help identify, modify, or eliminate the process which is not able to contribute to the value in the business. As the method of producing goods or services changes, the business process and business model also need to be changed accordingly to keep value creation. A typical example is information technology applied in business domain. The value in information technology is demonstrated in the next section.

2.2.3 Value in Information Technology

In the traditional corporate era, factory's goods are the primary products in exchange. In the information era, information technology becomes an important good in our life because both individuals and organizations need information to make better and quicker decisions. In information systems domain, value in information technology (IT) has been demonstrated as IT's impact on an organizational performance in efficiency and

competition (Melville, 2004). With the IT revolution propagated around the world, organizations are able to create, share, and analyze information more efficiently in business activities than ever before and ultimately create value in the form of process improvements, profitability, consumer surplus, supply chains, or organizational innovation (Kohli & Grover, 2008).

To realize the value from IT, four attributes of information are critical, including intrinsic information, contextual information, representational information, and accessible information (Lee et al., 2002). The intrinsic information indicates the accuracy and validity of the information. The contextual information refers to the relevant, timely, complete, and appropriate information that enables to present the context. The representational information focuses on the interpretation of the information. In other words, the information should be easy to be interpreted, presented, understood, and manipulated. The accessible information emphasizes that the information can be obtained appropriately, securely, and timely.

Although the four attributes of information provide a good guideline for building information systems that enable to create valuable information, in most cases, to completely meet all the attributes or requirements of the information through IT is not feasible or necessary (Cook et al., 1998). Because information systems not only face different users who have various requirements but also need to compromise on functionality due to limited resources. In practice, people always try to find an appropriate balance between the value created by IT and the resources consumed by IT (Cook et al., 1998). For instance, a retailer information system might provide different users different information about the on-sale products due to the diverse requirements. The customers are

only able to obtain the information about the price, function, customer review of the products, while the managers are able to browse more detailed information such as inventory, size, supplier, and cost of the products. Moreover, less critical information such as customer review and product supplier might be unsynchronized due the tradeoff between the value of the information and limited IT resources.

2.2.4 Value in Software Engineering

Value in software engineering focuses on stakeholders' expectations. The goal of software engineering is to create products, services, and processes that add value. "To maximize the value, software engineering decisions at all levels can be optimized to meet or reconcile explicit objectives of the involved stakeholders, from marketing staff and business analysts to developers, architects, and quality experts, and from process and measurement experts to project managers and executives" (Biffl et al., 2006, p. ix). That is, the value of software engineering is to provide high-quality programs which enable to satisfy the involved stakeholders' requirements.

Differing from the business value which spreads in the nodes of the value chain, the value in software engineering is created in each phase of software development life cycle (Biffl et al., 2006). In the requirement design stage, requirements engineering needs to identify the valuable stakeholders and elicit their value proposition for the software (Wohlin & Aurum, 2006). If the requirements of the program are not collected sufficiently and completely, it might result in the requirement change in the following steps such as software design or software development and in turn raises huge unforeseen resource consumption rather than value creation.

The architecture stage is about making a decision on the fundamental software architecture which is costly to change once implemented. According to IEEE standard (ISO/IEC/IEEE, 2011), software architecture is defined as "the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution." In simple words, architecture is the foundation of software development. To add value in software engineering, a good software architecture has to reconcile the software objectives with achievable architectural solutions.

In the design and development stage, software design usually involves problem solving and planning a software solution including both the high-level design, architecture design, and the low-level design, component and algorithm design (Boehm, 2006). Following software design, software development is a process of writing and maintaining the source code. But in a broader sense, it could include all the activities from the conception of the desired software through the final manifestation of the software. To add value in software engineering, developers cannot only focus on their own tasks, but rather they should always make good decisions in connecting the feasible development tasks to software requirements as well as achievable architectural solutions (Boehm, 2006). In practice, failed software projects usually distort this connection which cannot create any value.

Testing is one of the most widely used approaches for verification and validation and involves monitoring whether the software satisfies its objectives (Wallace & Fujii, 1989). Value-based view considers that not all the potential testing deserves to be seen or treated equally because different testing might create different value. For example,

functional testing heavily involves the early and middle stage of software development life cycle, such as module coding and system integration. Compared with performance testing and security testing, functional testing is the fundamental means of verification and validation to the eventual operation of the application, which is then supplemented by other testing such as performance testing and security testing.

2.2.5 Value in Software Testing

Value in software testing is defined as ensuring that a software solution satisfies its objectives and organizing testing tasks to operate as an investment activity to optimize the software testing (Boehm & Huang, 2003). In a long time period, a large number of software testing tasks is treated equally important in practice, resulting in limited resources in software testing not being able to achieve its optimal goal. To that end, a value-based view of software testing (Biffl et al., 2006; Hass, 2014) emerged which provides an effective approach for differentiating the importance of software testing activities.

According to the study by Ramler et al. (2006), value-based software testing incorporates two dimensions: An internal dimension and an external dimension. The internal dimension of testing covers costs and benefits of testing. This dimension includes the test activities in a project which are handled by the test group. Compare to other software development activities, software testing is not able to directly create value. Rather, its value is realized from supporting the critical software development tasks. For instance, testing the function of user registration in a program presents the value creating from internal dimension. Registration cannot directly stimulate user increase, but if users

have troubles with registration when they use the program, the registration test thus has a prominent value.

The external dimension emphasizes the opportunities and risks for the future system that need to be addressed. Differing from internal dimension, external dimension focuses on people outside the range of test activities such as end users, who may directly raise the risks and opportunities for the software. Although software testing engineers and developers are the people who directly get the benefit from the software testing, the stakeholders who are not directly involved in the software testing still need to be considered (Ramler et al., 2006).

Additionally, to optimize the value of software testing, we also need to take execution time into account. Specifically, a test executed in an early stage of software development is much more valuable than a test executed in a late stage. Because the earlier the test being taken, the faster developers are able to find and fix the bugs, and in turn to avoid huge loss if the defects are found after release. However, in a software project, there are numerous tests need to be executed where testers are not able to implement all the tests in an early time. Therefore, aligning the internal and external stakeholders' expectations in software testing plays an important role in prioritizing the tests (Biffl, S., 2006; Boehm, 2006; Boehm & Huang, 2003).

To summarize, the value of software testing could be maximized when the tradeoff between benefit and cost generated for internal and external stakeholders is optimized, and the critical tests are executed timely.

2.2.6 Value in Software Test Cases

First, value could be derived from monetary or benefit creation in something exchanged. In software testing, all the necessary testing processes provide the information about the validity and variability of the software application and also assist the software testing engineers in identifying defects. As a part of software testing, designing test cases is about creating input and predicting output that enable to test the certain parts of the program (Hass, 2014). Therefore, software test cases create the value for the software testing as the application failure is prevented from getting into the production environment, which might lead to significant losses after the application release. In other word, test cases create value in preventing different risks which are not supposed to emerge in the program. For instance, if a program is designed to be executed in different operation systems (Windows, Mac OS, and Android), the test cases for testing the compatibility are much valuable in multiple operation systems circumstance than in a single operation system circumstance (Cohen et al., 2003). Furthermore, any process in software testing including test cases is not free of charge. All companies attempt to decrease cost (e.g., software testing cost, cost of creating test cases) and increase benefit (e.g., application is reliable and free of defects) when they develop a program. Therefore, the value of software test cases is derived from identifying software defects and in turn preventing software failures (i.e., preventing risk) while, as the exchange, certain amount of resource is consumed in test cases (Hass, 2014).

Second, value is only able to be applied to the relevant people or stakeholders who are using the program (Biffl et al., 2006; Boehm, 2006). Test cases are usually created by software testing engineers, but other people may also be involved in software testing such as users or business people in marketing department because a defective application might

neither satisfies users nor increases sales in a market especially in a fierce competition environment. In general, the stakeholders of software test cases should include not only developers and testers but also business and marketing analysts, managers, and ender users.

Last, value occurs when something has unique utility (New World Encyclopedia, 2016). Test cases are generated to fulfill different functional testing purposes. In other words, each test case has its unique utility in software testing. If multiple test cases serve for the same goal without substantial difference, only one of them could create value for the software testing and the rest of the test cases might only waste limited resources. For instance, to test whether a program is able to show properly the delivery rate as users input the weight of a package, the test cases might be constituted by three types of numbers for a package: the number below the minimum weight limit or above the maximum weight limit, the number within the weight range, and the number on the minimum or maximum weight limit. If no any other factors need to be considered, only one test case should be created from each type of numbers.

Given the analysis in nature of value from different perspectives, we conclude that the value of test cases should incorporate two core elements: the risks being intentionally avoided by different stakeholders in the test cases, and the cost of the test cases. This finding provides a direction in proposing the value function, but it is not sufficient to build a deliberate value function. To this end, we design a case study in the following sections to explore the detailed elements in the value of test cases as well as the mechanism underling the value function.

2.3 Nature of Cost in Software Test Cases

Any value created must be based on a certain amount of resource consumption, which is also known as cost. The cost could be tangible such as money consumption or intangible such as time consumption. However, some costs are be easier to measure than some other costs due to the pattern of the consumed resources (Cooper & Kaplan, 1988). For example, the personnel expenditure for software testing is a more feasible cost metric than time or other types of effort consumed in software testing.

Given above attributes, cost has been successfully applying in business where the boundary and measurement for cost are very clear. Cost not only helps managers understand where resource has been allocated but also contributes to financial report as well as other managerial reports in a company. Although cost is a key index in practice and widely applied in different areas, the cost paradigm in software testing is still not well developed as that in business domain. Specifically, there is little research that objectively demonstrates the way in which testing contributes to the overall value of software development process (Talby et al., 2006). As software testing is increasingly costly, building a solid cost paradigm in software testing becomes more critical.

To this end, Gillenson et al. (2020) built a cost function of test cases that enables to help software testers estimate the resource allocation when creating test cases. We use their cost function as a part of basis in this research for the following reasons. First, the cost function focuses on test cases which are critical and fundamental in software testing. In contrast, other prior research studies demonstrate software testing cost as a whole which is not able to decompose the value of test cases. Moreover, the function breaks down the test cases cost into four categories based on testing process and demonstrate the

relationships among the four costs. Last, the cost function is applicable in various software testing methodologies such as traditional water-fall software testing or agile software testing. This wide spectrum builds a concrete foundation for value function of test cases in this current research.

2.3.1 Cost Elements of Test Cases

Gillenson et al. (2020) identified four types of cost: preparation cost (prep cost), creation cost, run cost, and failure cost. Each type of cost is constituted by several basic costs. The prep cost and creation cost are the one-time cost because these two costs usually occur once when starting to create test cases. The run cost and failure cost are the repeating cost because they might occur several times as long as the cases are executed multiple times, especially when bugs are found and fixed in software testing process. The cost of test cases is graphically represented in Figure 2, which is adapted from Gillenson et al.'s (2020) study.

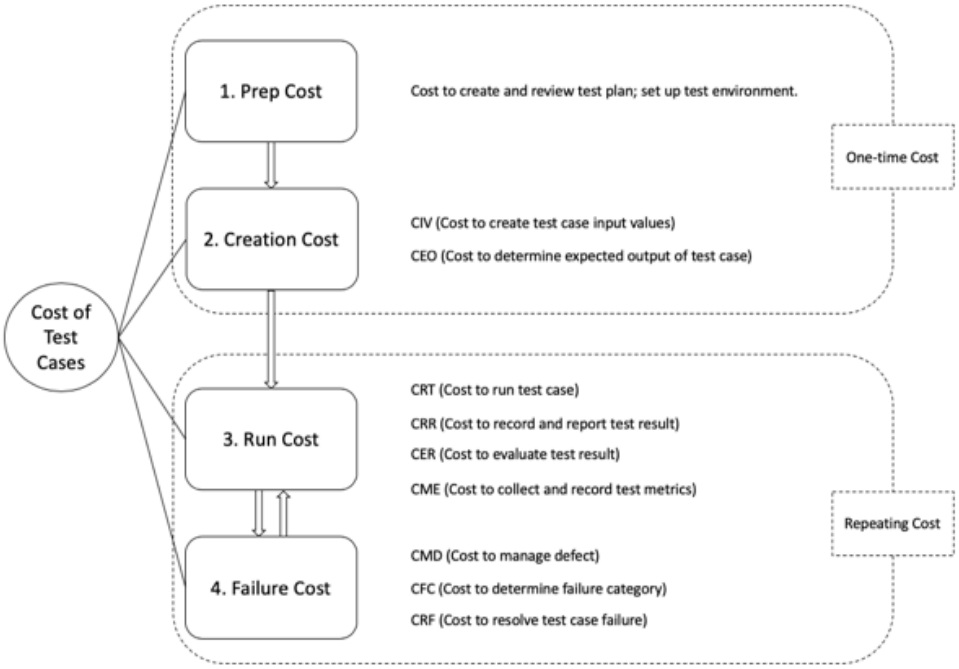


Figure 2. Model of Test Cases Cost

The *prep cost* is one-time cost to the whole testing effort. It occurs when creating and reviewing test plan, and setting up test environment where software and hardware are set up to execute test cases. When setting up the test environment, several challenges might render additional cost. For example, if test environment is located geographically apart, the test team and test assets may need more resources (e.g., time, people) in the coordination than in a local test environment. Moreover, complex test usually requires more complicated configuration in test environment, resulting in challenges to the test team (Gillenson et al., 2020).

The *creation cost* is one-time cost for an individual test case, which includes the cost to create input values (CIV) and the cost to determine the expected outcome of the testing process (CEO). Since test cases need to be very specific and cover all the possibility derived from a test scenario, input and output values might be a huge volume (Gillenson et al., 2020).

The *run cost* is a summation of repeating cost in executing test cases, which includes the cost to run the test case (CRT), the cost to record and report the results (CRR), the cost to evaluate the results (CER), and the cost to collect and record test metrics (CME) (Gillenson et al., 2020).

The *failure cost* is also a repeating cost but only when a defect is found. Failure cost is constituted by three basic costs. First, the cost to manage a defect (CMD), which is a cost that will always be charged to the testers. This defect management activity includes tracking the failure through assigning it to the responsible party for correction, making sure the correction has been completed, and reintroducing the test case into the mix.

Second, the cost to determine the failure category (CFC) should also be borne by the testers. There are four major failure categories: a code error, an error in calculating the expected output of the test case, a hardware or software problem with the test environment, or an error in the intended input values (derived from requirements) leading to an unintended negative test case. Third, the cost of resolving the test case failure (CRF) should be assigned to the party responsible for the error that caused the failure. A code error should certainly be charged to the developers. A problem with the test environment should be charged to the testers. Errors in calculating the test case input values or the expected output should be charged to whoever was responsible (Gillenson et al., 2020).

2.3.2 Cost Function of Test Cases

The cost function of test cases (Gillenson et al., 2020) incorporates all aforementioned costs. The pattern of the costs (i.e., one-time and repeating) is also presented in the function as follows:

$$\text{Cost of a test case} = \text{Prep Costs} + \text{CIV} + \text{CEO} + \sum_1^n(\,\text{CRT} + \text{CRR} + \text{CER} + \text{CME}\,) + \sum_0^a(\,\text{CMD} + \text{CFC} + \text{CRF}\,) \quad (1)$$

where n is the number of times the test case is run and a is the number of times the test case fails.

Notice that the upper limit *n* in the summation reflects the number of times the test case is run independently of any issue of test case failures. Also note that the lower limit of the second summation factor is *0* because some test cases may never produce a failure.

CHAPTER 3

RESEARCH METHODOLOGY

3.1 Research Process

Following the prior theoretical exploration of software test cases, we initiate this research regarding the importance and value of software test cases. The research process is based on the principle of the single case study research methodology (Yin, 2017) and the study is conducted in the process illustrated in Figure 3. In the initial stage, we introduce the practical and research issues in software testing that hinder organizations from optimally choosing effective software testing and allocating resources for software development. To address this complex issue, we form the research objectives and research questions to build a value function of a test case comprehensively representing the

importance of a test case, which is a critical part of software testing. In the middle stage, the nature of value, software testing, and test cases are theoretically explored to build the initial value function. Next, we collect the data regarding to the comments of the initial function through 27 in-depth interviews with software testing professionals and managerial staff who are working in a global Fortune 500 company and its American branches. Through analyzing the data, we finalize the value function and evaluate the revised value function in comparison with other popular estimation tools in the last stage.

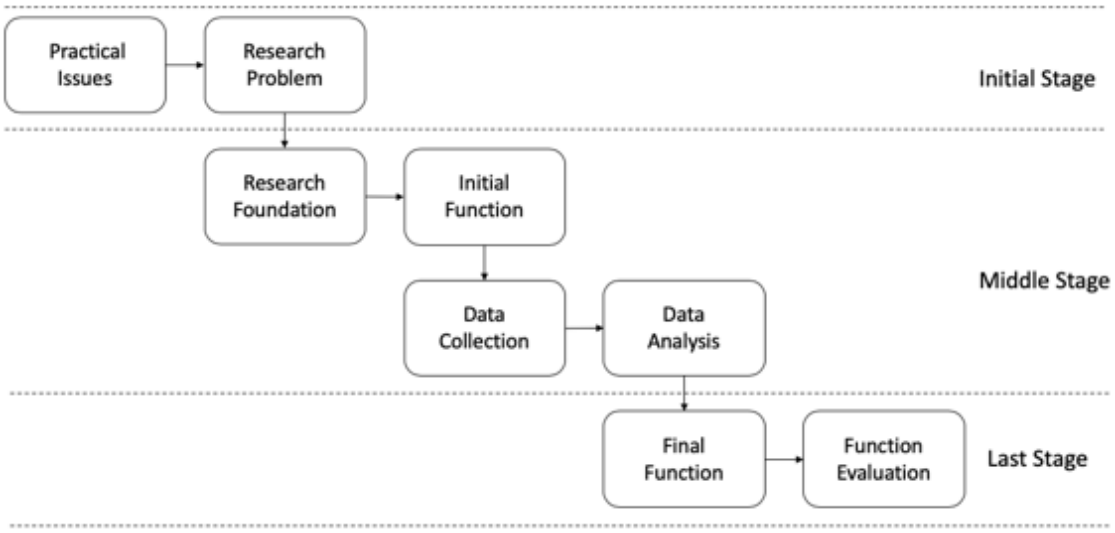


Figure 3. Research Process

3.2 Research Method

In this study, we choose case study research methodology (Yin, 2017) which is one of the widely accepted qualitative methods in social sciences, education, law, business, IS, as well as many other disciplines such as health and computer science. We select this method for two reasons. First, qualitative research methods are developed to help researchers understand people and the social and cultural contexts within which they live (Myers, 1997). In the software engineering field, many studies are related not only to

technical issues (e.g., algorithm) but also to non-technical issues (e.g., resource management) as well as to the intersection between the technical and non-technical aspects (e.g., database management). Since no one can directly and accurately calculate value and no unique standard of value of test case exists, estimating value of a test case is more complex than other software engineering issues. Considering this situation, we contend that qualitative method is more appropriate than quantitative approach which heavily relies on the findings from numerical analysis and statistics without adequate contextual exploration.

Second, action research, ethnography, grounded theory, and case study research are the primary qualitative research methods (Myers, 1997). Action research aims to contribute both to the practical concerns of people in an immediate problematic situation and to the goal of social science by joint collaboration within a mutually acceptable ethical framework (Rapoport, 1970). According to Myers's (1999) study, ethnography is the study that needs researchers to immerse themselves in the life of people they study and seek to place the phenomena studied in their social and cultural context. Ground theory is a research method that seeks to develop theory that is grounded in data systematically gathered and analyzed (Myers, 1997). The case study methodology, as defined by Yin (2017), investigates contemporary phenomenon in its real-world context when the boundaries between phenomenon and context may not be clearly evident (see Table 6).

In contrast with case study, we discovered that the former three research methods focus on either intensive and long-time observation of the phenomenon (e.g., action research and ethnography) or theoretical development (e.g., grounded theory). Considering our research objective and research condition constraints, we considered that

case study is more suitable to this study because this method enables to develop an in-depth description of the software testing context and explore the value function of a test case within an economic time period.

Table 6. Qualitative Research Methods in IS Domain

Method	Description	Reference
Action Research	Aim to contribute both to the practical concerns of people in an immediate problematic situation and to the goal of social science by joint collaboration within a mutually acceptable ethical framework.	(Rapoport, 1970)
Ethnography	Need researchers to immerse themselves in the life of people they study and seek to place the phenomena studied in their social and cultural context.	(Myers, 1999)
Grounded Theory	Seek to develop theory that is grounded in data systematically gathered and analyzed.	(Myers, 1997)
Case Study	Investigate contemporary phenomenon in its real-world context when the boundaries between phenomenon and context may not be clearly evident.	(Yin, 2017)

3.2.1 Data Collection

Initial value function of test cases is proposed before the data collection whose purpose is to get feedback on the initial value function. We conduct in-depth interviews over a three-month period with 27 software testing professionals and managerial staff from industry. Interview is a very common method in case study requiring researchers (1) to follow their own line of inquiry, as reflected by the case study protocol, and (2) to ask interview questions in an unbiased and fluid rather than rigid manner (Yin, 2017). In order to achieve a consistent and fluid line of inquiry and promote objective responses, interview questions were general and not related to specific test cases or circumstances. Each interview lasted about one hour and was guided by a documented and uniform

procedure (see Appendix A): introducing the initial function, explaining key function concepts, presenting the prepared questions, and providing time for ad-hoc questions that might arise in the process of the interview. All the interviews are conducted by two researchers while an interview note is created. To ensure the note is consistent with the respondent's original insight, each interview note is finalized in an interview summary as the respondent comments on the interview note via a follow-up email.

All the interviewees are working in a global Fortune 500 company and its American branches, which heavily rely on IT in its worldwide business. To promptly adapt continuous changing environment and fierce business competition, the company has more than 5,000 software developers and about 800 test engineers scattered at multiple locations internationally where most of the required systems are developed and tested.

In Marshall et al.'s (2013) research about the sample size of qualitative study in IS research, they found that single case studies should generally contain 15 to 30 interviews. As such, 27 interviewees were recruited in this study.

3.2.2 Data Coding and Analysis

A code in qualitative inquiry is most often a word or short phrase that symbolically assigns a summative, salient, essence-capturing, and/or evocative attribute for a portion of language-based or visual data (Saldaña, 2015). In order to systematically analyze interview data, we follow Saldaña's (2015) coding manual to generate a code book for clustering the interview transcripts which express similar topics. Specifically, the coding process is to create a short phrase to describe the main idea of similar comments. As the critical foundation of data analysis, the code book needs to be reliable and accurate to reflect the essence of the entire interview data. In raising the reliability and accuracy of the

code book, we generate two code books during the process: code book v1 (see Appendix C), the initial version, and code book v2 (see Appendix D), the final version.

First, to generate code book v1, two coders start coding the interview transcripts separately and generate two drafts of the code book based on their understanding of the same interview data. Since there is not a unified standard and different coders may make different judgements, the differences between the two drafts of the code book are expected. To reconcile the difference, a third coder playing the role of coordinator is invited to hold a meeting where the first two coders have to justify their codes if differences exist. Through the thorough discussion, the two coders, in most cases, can understand each other's judgements and achieve an agreement on the different codes. Otherwise, the third coder makes the final decision by listening to the two coders' discussion. Then, code book v1 (see Appendix C) is generated as the agreement has been reached among the coders.

Second, to generate code book v2, we make several revisions based on the code book v1 to facilitate data analysis. A code book usually needs to be adjusted in several rounds to reach a mature state where the entire data can be exhibited clearly and concisely (Saldaña, 2015). To improve the code book, we conduct code merging, for simplifying the structure of the code book, and code splitting, for separating multiple semantic meanings within a single code.

Code Merging: Code "litigation" and "globalization" as well as their corresponding interview transcripts (i.e., 5(6), 5(7)) in code book v1 are merged into code "External Risk." Code "Special Case" and its interview transcript (i.e., 1(4)) in code book v1 are merged into code "Amount of use." There is only one interview transcript

belonging to each code (i.e., “litigation”, “globalization”, and “Special Case”). To simplify the structure of the code book, they are merged into the existing codes which enable to cover the semantic meaning of the merged codes.

Code Splitting: Code “Unit Value” and its corresponding interview transcripts in code book v1 are split into four codes: “Unit Value,” “Regression Suite,” “Test Utility,” and “Dynamic Function.” When generating code book v1, we categorized the interview transcripts into code “Unit Value” if they could not be classified into other codes. “Unit Value” becomes a code which incorporates broad perspectives regrading value of a test case. This results in a problem that the diverse interview transcripts in the “Unit Value” cannot be extracted from a unique angle or discussed adequately. To separate multiple semantic meanings within a code, we conduct code splitting. The four resulting codes and their corresponding interview transcripts can be found in Appendix D.

Table 7 presents coding examples with the interview transcripts in code book v2. Be aware that coding is not a precise science. Therefore, different people may come up with different code upon the same material. The rule of thumb is that a good code is usually able to summarize, distill, or condense data (Saldaña, 2015).

Table 7. Coding Example

Excerpt of Interview Transcript	Code
“Maintaining the line items of Technical Risk in a table rather than putting each of them separately in the unit value function is a good idea.”	Technical Risk
“The amount of use is a very critical factor. Multiplying external risk by it might be insufficient to elaborate its important role. Multiplying the entire set of risk factors is an option for this point.”	Amount of Use

"The value of a test case is directly related to the amount of revenue that the software under test is likely to generate. The more revenue the software is likely to generate, the more value the test case possesses. This could be an additional factor in the unit value function."	Revenue Generation

Third, condensed viewpoints are distilled from the converged interview transcripts in code book v2. Those distilled practical comments constitute the guideline of revising the value function. To present a clear analysis, the codes in code book v2 are classified into two groups: function codes and non-function codes. If the codes are substantially supported by their interview transcripts to be added in the value function, the code will be classified into the function codes group. If the codes are not incorporated in the value function based on the interview transcripts and justification, the code will be classified into the non-function code group.

In the following chapters, we adopt the function codes into the value function and adjust their mathematical expression of the factors (i.e., function codes) as well as items of the factors. Then, the final value function is developed.

CHAPTER 4

INITIAL VALUE FUNCTION

4.1 Process of Proposing Initial Value Function

To incorporate knowledge from both industry and academia, we propose the initial value function as the process exhibited in Figure 3. First, we consult a software testing expert (key informant) about the primary factors in influencing a test case. Based on the real environment of software testing, he contends that risks mitigated by a test case (positive influence) and cost occurred for the corresponding test case (negative influence) contribute the value of a functional test case. The expert is a veteran in software testing and has been working as a senior director for more than 10 years in the software testing department of a global fortune 500 company where we recruit the interviewees in the data collection process. Next, having the knowledge from the key informant, we further explore the nature of value, risk, and cost associated with test cases from the existing

research. The findings have been illustrated in the prior section. Last, we reconcile the knowledge from the two sources and then propose the initial value function.

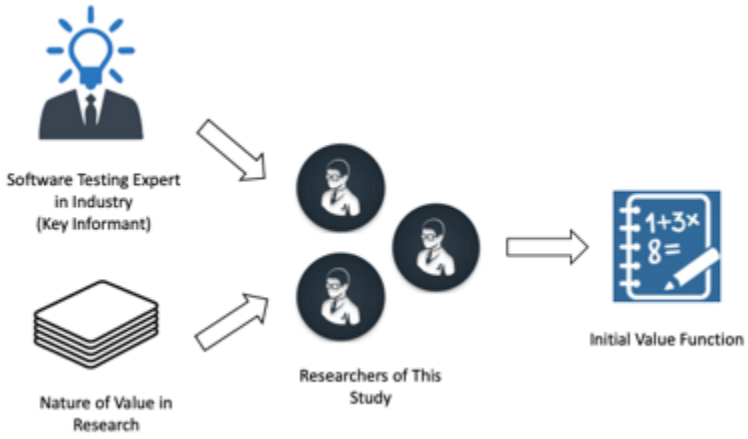


Figure 4. Process of Proposing Initial Value Function

4.2 Factors in Initial Value Function

4.2.1 Risk Factors

Based on our exploration of the nature of value, software testing, and test case in the research foundation, we consider that the initial testing value function would contain two specific kinds of factors that influence the level of value in the process. One category is the expected return (or, benefit) which provides a positive influence on the function – it is able to increase value. In practice, there are countless factors that could add value. As Hass (2014) aptly notes, the best tests reduce the risk of defects remaining in the product when it is released to the customer. Hence, a good way of evaluating positive factors in a proposed testing value function is to identify the capability such factors have for reducing the likelihood of defects escaping notice in testing. Fewer defects means higher value, essentially, and this is a risk-reduction calculation of value.

In terms of characterizing types of risk in software testing, there is currently no uniform classification scheme, although Hass (2014) has offered a classification of four risk types: business risk, processes risk, project risk, and product risk. From the company-wide perspective, risks can be categorized as strategic, compliance-related, financial, operational, and reputational (Griffin, 2019). Combining the key informant's viewpoints and the nature of the value in test cases, we contend risks in software testing (when the risk event is the unintended release of a defect to the customer) are either indirect risks (comprised of business and operational issues without directly implicating the testing process) or testing-specific and direct risks, characterized as "technical risk".

Business & operational risk relates to the importance of the software to the operation, integrity, or financial stability of the company. We think of these indirect risks as either *Internal Risk* (IR), which refers to risks arising from events taking place within the organization and *External Risk* (ER), which refers to the risk arising from the events taking place outside of the organization. To aid the reader in conceptualizing the risk factors in a value-laden framework, Table 8 provides a visual characterization of the risk factors juxtaposed against specific operational instances that may manifest in company operation. For internal risk, we consider that executive pressure within the company and either resource deficiencies or poor organization of resources are the two primary items impacting the goodness of testing results. Externally, crucial impacts are failure in production, relevant regulations, and fierceness of competition.

Technical risk (TR) begins with the complexity of the software. A more complex piece of software is inherently riskier in production and testing than a less complex piece of software. Furthermore, a larger program or portion thereof is riskier than a smaller

program simply because the larger program presents more opportunities for error. Technical risk also involves factors related to programmers, test engineers, and their tools. More experienced programmers present less risk to the finished product than less experienced programmers do, for example. New testing technologies in use are riskier than established technologies. To that end, we have derived primary components of technical risk listed in Table 8: complexity issues, technology issues, requirements issues, personnel issues, dependency issues, previous testing issues, and test environment issues.

Table 8. Factors in Initial Value Function

Category		Primary Item
Business & Operational Risk	Internal Risk (IR)	Executive pressure within the company
		Deficiency or poor organization of resources
	External Risk (ER)	Crucial impacts of failure in production
		Relevant regulations
		Fierce competition
Amount of Use (AU)		The relative amount of use in production
Technical Risk (TR)		Complexity issues
		Technology issues
		Requirements issues
		Personnel issues
		Dependency issues
		Previous testing issues
		Test environment issues
Cost of a Test Case (CTC)		Preparation costs
		Creation costs
		Run costs
		Failure costs

4.2.2 Use Factors

Besides the two main factors discussed in regard to our proposed value function, another special factor related to external risk is the amount of use (AU) of the software under test. A frequently used piece of software, whether it is a full application or a feature of an application, is inherently riskier than an infrequently used piece of software. This

may seem counter intuitive, until one considers that external risk increases for widely used applications that experience failures, as the impact of the failure is more widely distributed and more costly to correct. Therefore, we combine external risk and amount of use together in the initial function to demonstrate this potential negative impact on value. In sum, we consider the business and operational risks, technical risks, unit costs, as well as amount of use as the primary factors to evaluate the value of a test case (see Table 8).

4.2.3 Cost Factors

The other major factor is related to costs that negatively impact value. Specifically, value decreases with increased costs, and, all things being equal, decision makers prefer to choose the functional test cases that are less costly in order to preserve value in the process. Therefore, in the initial unit value function, cost of a test case (CTC) becomes a key factor. When estimating CTC, one must take into account the costs of test case creation, the costs of running a case, the costs of determining success or failure for the case, and, possibly, the related costs of using it to fix a subsequently uncovered defect in the code. A detailed explication of the cost structure for test cases can be found in Gillenson et al. (2020). The value function, derived from these costs, is characterized by the factors represented in Table 9.

Table 9. Factors in Cost Function

Category	Code	Items
Prep Costs		Cost to create and review test plan; set up test environment
Creation Costs	CIV	Cost to create the test case input values
	CEO	Cost to determine the expected output of the test case
Run Costs	CRT	Cost to run the test case
	CRR	Cost to record and report the test results
	CER	Cost to evaluate the test results
	CME	Cost to collect and record test metrics, if required

Failure Costs	CMD	Cost to manage a defect
	CFC	Cost to determine failure category
	CRF	Cost to resolve test case failure

4.3 Factor Relationships in Initial Value Function

In the previous section, we introduced IR, ER and TR as the three key elements which can increase the unit value of a test case when the risks are properly managed. Each of those risks, when judiciously resolved in the software testing process, act to decrease the probability of software failure and hence increases the value of the test case. On the other hand, any test case will consume resources such as funds and personnel, and this is also represented in the unit cost calculation. It is important to recognize that different test cases present different IR, ER, TR and CTC elements. In order to adjust the diverse influence of the four factors in the function, weighting is applied to each factor in the calculation. The initial function of unit value of a test case is represented as follows:

$$\text{Value of a Test Case} = IR*w_1 + (ER*w_2) * (AU*w_3) + TR*w_4 - CTC*w_5 \quad (2)$$

where w_i for $i = 1, \ldots, n$ is the weight of each factor.

In this function, there are several points should be noted. First, value is the *relative* worth of expending additional resources to add the test case under consideration to the testing effort. Also, software under test (SUT) can be a feature of an application component, an entire application, or even a collection of integrated applications. IR, ER, AU, and TR are all based on the SUT while CTC is based on the test case under consideration. Mathematically, all five of the factors are scaled from 1-n in whole numbers; factor weights can range from 0, upwards, and can include fractional

components but the sum of the five weights should equal one. This could present the different impacts from the five factors when estimating different test cases. The point of multiplying ER and AU arises from the consideration that a major failure in a heavily used application should likely have synergistically deleterious effects; further refinement can be made with the weighting coefficients w_2 and w_3. Lastly, for CTC the entire cost function should be used, because an estimate of the eventual run costs and failure costs based on history is an important component in the calculation.

CHAPTER 5

DATA ANALYSIS

5.1 Data Description

Although the initial value function was derived from theoretical foundations, a gap might exist between the theoretical insights and the practical situation. Therefore, we interview and analyze the feedback of initial value function from industry experts. In the interview process, we primarily collect two types of information from interviewees: What are the necessary factors of the value function? What are the relationships among the factors? Following the aforementioned coding and analyzing process, 16 codes (internal risk, external risk, technical risk, amount of use, code coverage, test frequency, weight, unit cost, dynamic function, unit value, test utility, regression suite, revenue generation,

simplicity, priority, and ROI) are identified in code book v2 (see Appendix D). Figure 5 exhibits the frequency of code occurrence in the interview data. The blue columns and orange columns respectively represent function code (factors included in the final function), and non-function code (factors excluded from the final function). Note that different factor names are used in the value function for some function codes: external risk, code coverage, and dynamic function. The explanation can be found in the following code analysis.

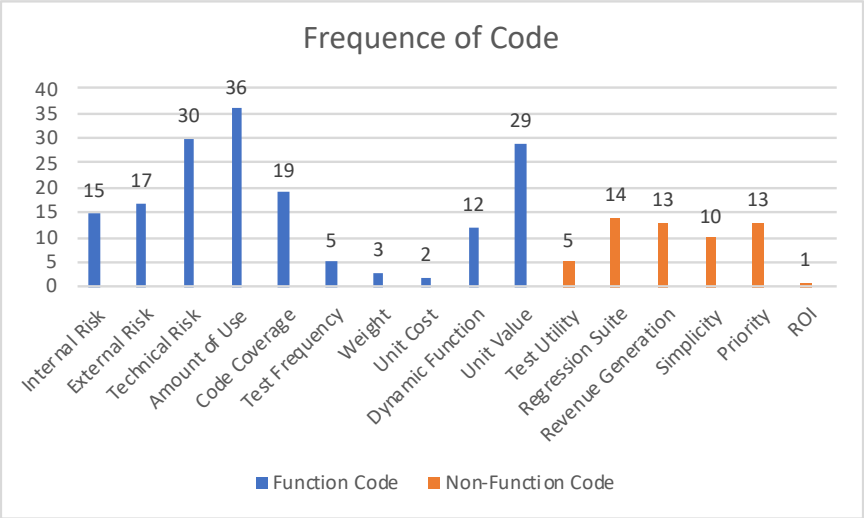


Figure 5. Frequency of Code

5.2 Findings from Function Codes

Function codes are the factors incorporated in the final value function. In the previous section of proposing the initial value function, internal risk, external risk, amount of use, technical risk, weight, cost of a test case, value of a test case are incorporated in the initial function. In the following section, those existing codes as well as other new codes (function coverage, test frequency, execution value) are discussed and justified based on the knowledge obtained from the code book v2 (see Appendix D). The interview comments of the code usually provide three types of information of the code: (1) whether

the code is closely related to the value of a test case. In other words, if the code is appropriate to be included in the value function; (2) the definition of the code, including what lower level items constitute the code; and (3) the relationship with other factors in the value function. The function code analysis will be developed from these three perspectives.

5.2.1 Internal Risk

Internal risk (IR) is the risk arising from the events which take place within the organization but are not related to technical operations, which could result in potential losses that could be eliminated or eased by the testing. There are 15 interview transcripts coded as internal risk. Three critical points of emphasis requisite to justify internal risk: the risk derives from internal operations of the organization, the risk may directly or indirectly lead to near-term loss, and the risk could be resolved by testing (Iversen et al., 2004).

From the interviews, most comments focus on the nature of internal risk and the executive pressure and poor organization of resources are reflected as the main sources arising the internal risk to applications. Some typical comments are suggested by interviewees:

> *IR is assigned at the executive level. Testers on the front lines, normally have to follow the high level managers' directions regarding risk factors.*
>
> *Executive pressure in Internal Risk (IR) can go two ways. "Do it well," in which case the unit value of a test case should go up.*

Executive pressure may be about quality or speed of development, or both.

Based on the consensus of the interview data, two pervasive administrative issues -- executive pressure within the company and deficient or poor organization of resources -- are considered as the main indicators of internal risk.

5.2.2 Production Risk

Production risk refers to the risks arising from application failure which is not related to technical operations. Losses arising from production risks could be eliminated or eased by testing. In the initial function, production risk is mainly represented by external risk (ER) which only arises from events outside the firm (Hoodat & Rashidi, 2009). However, some applications are developed for internal use rather than for public or external use. The external risk cannot entirely reflect this special case. Thus, we use production risk instead of external risk in the final value function. Seventeen interview transcripts are coded in this category.

Similar to internal risk, production risk is constituted by three items which are identified from the interview data. They are impact of failure in production, relevant regulations from law and convention, and fierce competition. Some interviewees stated:

ER is based on the probability of the software failing in production.

The ER could be affected by the impact from social media that may influence the potential customers' judgement.

External risk increases if the software is intended to be used in many countries on a global basis. Therefore, test cases that test this software are more valuable.

Some interviewees mentioned, external risk/production risk and internal risk can dynamically interact with each other. For example, as marketplace competition subsides, internal executive pressure, which is part of IR, might be eased accordingly. However, we contend that, in most circumstance, the two risks are distinct and should be identified as they influence the value differently. Therefore, we add the production risk in the value function.

5.2.3 Technical Risk

Technical risk (TR) is derived from technical issues regarding the application which could not be included in the business and operational risk factors (i.e., internal risk and production risk). In this case, IR, PR, and the potential loss TR could be eliminated or eased by testing. Thirty interview transcripts are coded in this category.

In the interview data, those test practitioners point out TR may arise due to different perspectives. Some interviewees suggested:

All the items listed in TR help people from diverse perspectives evaluate the confidence of the testing staff for completing a test case well. Different testing groups in different situations, however, perceive different TR line items to be more or less important. Thus, the TR line items should be weighted separately to allow for the needed diversity.

> *Complexity and dependency of the software are more critical than the other items of TR. However, the critical level of the items could vary in different situations, so having TR line items with separate weights is a good solution.*

As such, all seven TR items in the initial value function are retained in the revised function based upon comments from the interviewees. Complexity issues (CI) refers to TR arising from a complex application (Gefen et al., 2008). Technology issues (TI) refers to TR arising from technology problems which cannot be well resolved when developing the application (Hoodat & Rashidi, 2009). Requirements issues (RI) involves TR arising from requirement management where users' expectations of the application cannot be fully satisfied or are out of control (Iversen et al., 2004). Personnel issues (PI) involves TR arising from the application which was not developed by experienced developers (Hoodat & Rashidi, 2009). Dependency issues (DI) involves TR arising from the application which couples with other applications. Previous testing issues (PTI) refers to TR arising from a poor historical test record of the application. Test environment issues (TEI) refers to TR arising from the test environment generated for the application. Moreover, an interviewee suggested:

> *The function also should be considered from vendor group perspective rather than only from testing group perspective.*

Therefore, vender issue (VI) is added to the function as a new item which represents issues caused by software testing vendors in regard to the test case (Gefen et al., 2008) (see Table 10).

Table 10. Items of Technical Risk

Item	Weight
Complexity issues (CI)	w_1
Technology issues (TI)	w_2
Requirements issues (RI)	w_3
Personnel issues (PI)	w_4
Dependency issues (DI)	w_5
Previous testing issues (PTI)	w_6
Vendor Issues (VI)	w_7
Test environment issues (TEI)	w_8

Since those items may have different impact as the test context varies, it is necessary to add weight for adjusting the variance as an interviewee suggested:

> *It is better to break down the TR into several subcategories, each of which has its own weight.*

So, the eight items of TR with separate weights are listed in the following revision of the function:

$$TR = CI * w_1 + TI * w_2 + RI * w_3 + PI * w_4 + DI * w_5 + PTI * w_6 + VI * w_7 + TEI * w_8 \qquad (3)$$

where w_i for $i = 1, \ldots, n$ is the weight of each factor.

5.2.4 Amount of Use

Amount of use (AU) is the relative index indicating the extent to which an application is being used. Thirty-six interview transcripts are coded in this category. Since AU is a relative index and may vary in different industries, this factor needs to be independently estimated by experts for different contexts. For example, AU should vary

widely between the context of a globally utilized social networking application and a local weather report application as well as a private application. One interviewee suggested:

> *The amount of use is a very critical factor. Multiplying external risk by it might be insufficient to elaborate its important role. Multiplying the entire set of risk factors is an option for this point.*

Some other interviewees argued:

> *ER multiplied by AU is preferable, since AU is much more directly correlated with ER than with the remaining factors. If the software fails, you are going to lose revenue or customers.*
>
> *Uncertain about whether AU should be expanded to multiply more risk factors or not.*

Although the point of views regarding AU is discrete, we think AU is dependent upon both the software customers who come from outside of the organization, the software users within the organization, and technical risk which shall impact the application development. For this reason, we contend that AU is a significant factor directly influencing both IR, PR and TR. So, the three risks are multiplied by AU to present this relationship in the final function.

5.2.5 Function Coverage

Function coverage (FC) is a concept of testing which is not included in the initial value function. We identified FC when analyzing the interview data of code coverage

(CC), which is totally different from the approach of getting PR. FC refers to how many features of a program are covered by a given test case. A program usually consists of multiple features which are represented by resource codes. Therefore, a test case with high function coverage indicates a higher capability of the test case in testing the features as well as the corresponding resource code also known as code coverage. Consequently, such a test case would decrease the number of test cases needed during the testing process (Lin et al., 2012). Nineteen interview transcripts are coded in this category.

In interview data, both function coverage and code coverage are supported. An interviewee stated:

> *The value of a test case should be based on the function points (i.e., requirements) instead of the amount of code or of specific parts of the code covered.*

Some other interviewees believe that code coverage is also a critical factor influencing the value. Some interviewees suggested:

> *A test case that covers more code is more valuable than a test case that covers less code but helps find the location of a defect.*

> *Code coverage (and therefore application features implemented) should be considered when evaluating the value of a test case. A test case that tests more of the code (and by extension more of the application features) has a higher value than test cases that cover less code.*

Although function coverage and code coverage are supported in the relationship of test case value, we find function coverage positively influences code coverage but it fails in the reverse relationship. In other words, higher function coverage leads to higher code coverage, but higher coder coverage would not necessarily result in higher function coverage. Others being equal, generating fewer test cases which are able to cover all the functions that are supposed to be tested can save more resources and thus create value. As such, we decide to incorporate function coverage but not code coverage in the value function.

5.2.6 Test Frequency

Test frequency (TF) is another critical standard which can be utilized to evaluate value of a test case. Five interview transcripts are coded in this category. High usage frequency of a test case always presents greater value and higher priority compared to test cases with low usage frequency (Lin et al., 2012). This viewpoint is strongly supported in practice. Some interviewees pointed out:

> *The more places in the development cycle a test case is used, the more valuable it is.*

> *Repeatability, meaning whether a test case can be used across different regions, devices, or platforms, is a factor in the value of a test case. High repeatability indicates high value of a test case.*

To save time and effort, testers usually prefer to use a test case in a regression test suite rather than writing a new test case. As test frequency rises, the value of a test case increases. Moreover, as we discussed in the prior section, test case value can be reflected

by function coverage. Therefore, the test frequency of a test case can enhance the value of the test case in terms of function coverage. Thus, we multiply function coverage by test frequency in the function to reflect the close relationship of the two factors.

5.2.7 Cost of a Test Case

In the prior study, Gillenson et al. (2020) have developed a cost function of a test case and considered the cost as an inherent element of the value of a test case. Since this study focuses on exploring the value of a test case, we directly adopt the cost function as part of the value function and do not intend to revise this factor in the following study. So, we treated this factor differently from other factors during the interview process. The structure of cost of a test case was introduced to help interviewees understand the initial value function, but we did not ask interviewees to make comments on revising the cost of a test case (CTC).

5.2.8 Execution Value

Execution value (EV) refers to the value of a test case after being executed. This new factor is derived from analyzing the interview data of "dynamic function" which is constituted by twelve interview transcripts. From the data, a lot of interviewees believe the value of a test case is not static. It may change as the impact of the test case is different from the expectation which was supposed to be after the test case has been used. Some interviewees stated:

> *Recursive use of the value function – do you find greater value after you start running the test case because, for example, it finds a lot of defects.*

The value of a test case may change as you use it in testing. There is a scenario that is pervasive in practice. A test case that was considered to be low value in the initial stage may increase in value due to more defects being detected after the test case execution.

Although the value function is intended to reveal a test case value and help testers in optimizing testing resource allocation on different test cases before execution, reconsidering the value of a test case in real-time is necessary because this enables us to reflect ex-post evaluation of the test case. To make the final function dynamic, we add EV in the function, which could be derived from the value estimation after the test case execution. Then the function incorporates two pieces of information reflecting the value. One piece is EV reflecting the value adjustment after the test case execution. Another piece is the remaining part of the value function reflecting the value estimation before the test case execution. If the test case has never been executed, the recursive value of the test case should be zero. To summarize, the value of a test case could become dynamic as EV is incorporated in the function where the impact after the test case being executed has been taken into account.

5.2.9 Value of a Test Case

Value of a test case (VTC) is a fundamental concept to the entire notion of test case valuation. Twenty-nine interview transcripts are coded in this category where interviewees provide ample insights on understanding and deconstructing the value of a test case. In order to explicitly demonstrate the value of a test case, it was necessary to clarify the assumptions of the study for our respondents during the interview process. In

our study we were only focusing on the value of functional test cases. In other words, other types of test cases such as performance test cases were specifically not under consideration.

Given the assumption, all the interviewees addressed their insight based on the initial function. Subsequent data suggests the value of a test case is a comprehensive concept that presents the predicted return projected by the tested application, potential direct or indirect loss decreased by the tested application, the effectiveness and efficiency of the test case, and the cost of a test case. All the above function factors except the cost factor positively impact on the value of a test case. To summarize, internal risk, production risk, technical risk, and amount of use are associated with the application under test where indirectly influence the value of a test case. In contrast, function coverage, test frequency, and cost of a test case directly impact the value of the test case.

5.3 Findings from Non-Function Codes

Non-Function codes are the factors excluded from the final value function. In this section, the comments regarding the non-function codes (regression suite, test utility, revenue generation, simplicity, priority, and ROI) are discussed and justified based on the knowledge obtained from code book v2 (see Appendix D).

5.3.1 Regression Suite

Regression suite is widely used in the process of software testing. To save time and effort in generating test cases, testers usually add the test cases which may be repeatedly executed in a file known as a regression suite (Lin et al., 2012). The regression suite can be used to thoroughly test the application rather than running each test case

separately. When an application is modified, it is convenient to test it with regression suite. There are fourteen interview transcripts coded in this category, and some interviewees argued that:

> *A test case in a regression suite is more valuable than one that is not.*
>
> *The value of a test case increases somewhat if it is added to a regression suite.*

However, some interviewees suggested that:

> *The value of a test case is determined up front and a high-value test case is added into the regression suite. Value is not determined by the decision of whether or not to add it to the regression suite.*

Considering that the test frequency is the factor reflecting the extent to which testers use the test case, we exclude the regression suite from the final value function.

5.3.2 Test Utility

Test utility (TU) indicates the capability of the test case in defect finding. There are five interview transcripts coded in this category. A debate arising from the interview data is whether defects found by the test case could increase the value of the test case. Some interviewees argued:

> *Quality of a test case is important and is based on the number of defects found by it.*

The value of a test case increases if it detects defects in risky code.

However, some interviewees have an opposite opinion regarding the test utility. They stated:

A test case that finds no defect is just as valuable as one that finds defects. This has no effect on the value of a test case.

Differing from the prior statements focusing on error-finding, this argument emphasizes the importance of information which is obtained after executing the test case. Regardless of whether any defect has been identified, the test case is still worthy to be considered as long as the test case is responsible to fulfill the test objectives.

With the discussion, we contend that test utility is an inherent element of the value of a test case. The value of a test case is reflected by the information/result after the test case execution regardless of whether any defects are found or not. Specifically, if defects are found by the test case, it indicates that the test case is useful in checking the application. If no any defects are found, it may imply good quality of the application which is also a valuable piece of information for the testers. Therefore, whether a test case is able to test vast features of the application becomes more important. Since function coverage plays a similar role in the value function, we do not add test utility as a new factor in the value function.

5.3.3 Revenue Generation

There are thirteen interview transcripts coded in this category where several respondents had strong views about revenue generation as a possible factor in the value

function. The concern for revenue in the value function spanned both business management roles and technology development roles, alike. Generally speaking, with regard to for-profit organizations, the most important goal of operation is revenue generation. In such contexts, where revenue generation underlying the profit motive is a key concern, testing as a necessary part of application development would also have to create some direct or indirect revenue stream to justify its own value as an organizational function. An interview respondent said:

> *The value of a test case is directly related to the amount of revenue that the software under test is likely to generate. The more revenue the software is likely to generate, the more value the test case possesses.*

A different view suggests that revenue generation implies a degree of internal risk, given the high visibly of the revenue production process for executive oversight and competitive capabilities. And not all the software product is used to serve or sell to public. An interview respondent pointed out:

> *The business value of an application is the key issue and is more important than the revenue it brings in. Some applications are internal and do not bring in revenue.*

An interview respondent also argued:

> *The unit value of a test case depends on the business value of the software. Generating revenue is only part of the value of the software along with other business "options."*

Given the comments, we find revenue generation is a very general factors associating with various factors. Adding the revenue generation in the value function may arise the confusion in understanding the value of the test case. Therefore, we think that including revenue generation as a new factor in the final value function is not an appropriate choice.

5.3.4 Simplicity

Simplicity discusses simplifying either the construct of the value function or the structure of the factors in the value function. There are ten interview transcripts coded in this category where most test practitioners expressed their preference of having a simple value function. Because keep the function simple could be easily understood and also calculated. For example, some interviewees suggested:

> *Do not promote any line items in the table into the main function. It would get too complicated.*
>
> *Keeping function simple would help people readily comprehend the meaning of the function. Do not promote table rows into the function.*
>
> *To keep the function simple, expand out TR and UC in accompanying tables.*

Considering the application and interpretation of the value function, we keep the value function in a simple format, where detailed items of the factors (e.g., Technical risk is constituted by eight items) are not listed in the value function. Also, no extra factors need to be added in the value function regarding the viewpoint of simplicity.

5.3.5 Priority

The notion of priority is best expressed in the operational wisdom. Other conditions being equal, test cases with high priority always have a higher value than those with low priority. There are thirteen interview transcripts coded in this category where a sharp dichotomy among respondents arose as to what priority implies in practical terms.

One view is that the priority given a particular case should be a separate factor in the function. Suppose that there are two test cases developed for assessing the same financial calculation application, one test case testing an ordering function and the other testing a character-display function. It seems clear that the ordering function would have a higher priority than the character display function because if the former malfunctions, the critical financial calculations might be flawed. Compared with the test case for character-display functions, in which accurate calculations might simply be displayed poorly, the test case for the ordering function would appear to have the higher value between the two. As an interview respondent argued:

> *The priority of a test case is influenced by how critical the application feature is that the software is implementing, such as handling customer complaints, dealing with cutting edge technologies being used by competitors, etc.*

A contrasting view is that priority is already embedded in internal and production risk factors and would be hard to differentiate as a separate factor on its own. For example, the releasing deadline for an application pertains to internal risk, applications designed for handling customer complaints imply production risk, and dealing with cutting edge

technologies being used by competitors might suggest both internal and production risk factors. An interview respondent stated:

> *Priority always accompanies risk, especially in internal risk and external risk. It is hard to list priority separately. The standards of priority vary from one case to another. Sometimes satisfying customers, which is an element of external risk, is prioritized. In some other situations, release time, which is an element of internal risk, is considered the most critical element.*

The point of contention among respondents is that priority could be taken into account at much more than its normal impact if it was listed in the value function as a separate factor.

Considering that priority somehow plays roles in IR, PR and TR, setting up priority as a separate factor might cause confusion. Because separate setting raises an issue that priority is closely related to not only the value of a specific test case but also to the value of an entire application in which the case resides. To that end, we opt not to institute a new factor for priority in the value function.

5.3.6 ROI

ROI (return on investment) is one performance measure widely used for evaluating business projects which shares some features with and bears some similarity to the value function. In Phillips' (1997) study, ROI is defined as a percentage figure, arising from net program benefits divided by program costs, with net program benefits represented by total

program benefits minus costs. There is only one interview transcript coded in this category, suggesting that ROI could be a candidate tool for estimating the value of functional test cases.

From the functional calculations underlying both value function and ROI, we find several distinct differences. The value function is constituted not only by revenue/benefit factors but also considers the impact of risks, utility and use frequency factors. In contrast, ROI does not include these factors, and, accordingly, is not able to wholly present the value of test cases. To that end, the value function is much more comprehensive than ROI. Despite the more complex functional form, the calculation of unit values is easily programmed for processing by computer. Interestingly, the value function organizes various factors with different characteristics at the unitary level. In other words, all the factors are represented by their relative score and weight. In contrast, only the factors measured by monetary units are included in ROI. Lastly, the value function is able to present the scale of impact on value, whereas ROI is a basic ratio that can compare any project or program, regardless of size. This potentially leads to misdirection when the easily foreseeable outcome of a very small test case with extremely high ROI results in preferential choice, in contrast to a very large test case with relative lower ROI. Therefore, we think the ROI is neither an appropriate method to evaluate the value of a test case nor a good factor for being included in the value function.

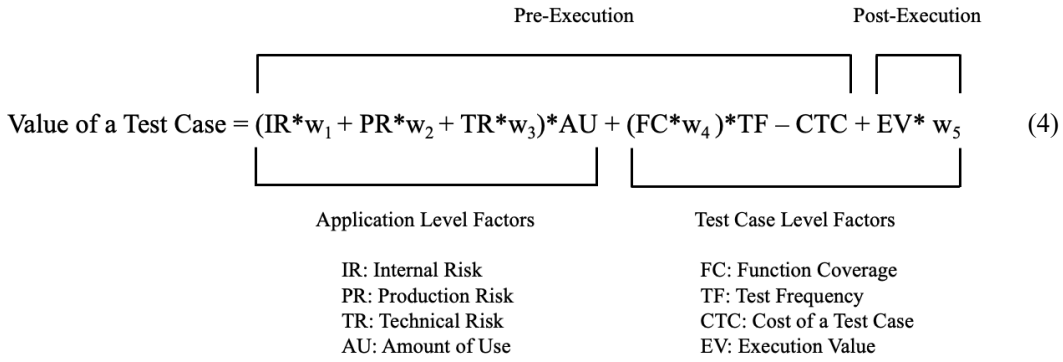
CHAPTER 6

FINAL VALUE FUNCTION

6.1 Final Value Function

Based on the previous discussion of the new factors and existing factors, the final value function is finalized and the key factors in the final function are listed in Table 11. Specifically, the value of a test case is a relative value that is comprised of a series of positive impact factors, negative factors, and an ad hoc factor in the function. Positive factors are constituted by internal risk, production risk, technical risk, amount of use, function coverage, test frequency, stimulating the VTC increase as those factors' score increasing. Among the positive factors, internal risk, production risk, and technical risk are associated with an application. The three application-level risks would increase in

importance as the use of the application increases. So, internal risk, production risk, and technical risk are multiplied by the amount of use in the function. Similar to the test-case level factors, function coverage is multiplied by test frequency. The cost of a test case is the negative impact factor, which leads the value of a test case to decrease as the cost score goes up. The ad hoc factor is execution value, adjusting the value of the test case after the test case execution. Given the prior findings, the final value function of test cases is:



where w_i for i = 1, …, n is the weight of each factors.

Table 11. Factors in Final Value Function

Category			Primary Items
Application Level	Business & Operational Risk	Internal Risk (IR)	Executive pressure within the company
			Deficiency or poor organization of resources
		Production Risk (PR)*	Crucial impacts of failure in production*
			Relevant regulations from law and convention*
			Fierce competition
	Technical Risk (TR)		Complexity issues
			Technology issues
			Requirements issues
			Personnel issues
			Dependency issues
			Previous testing issues

		Vendor Issues†
		Test environment issues
	Amount of Use (AU)	The relative amount of use in production
Test Case Level	Function Coverage (FC)†	The relative amount of function of the program being tested
	Test Frequency (TF)†	The general use frequency of the test case
	Cost of a Test Case (CTC)	Preparation Costs
		Creation Costs
		Run Costs
		Failure Costs
	Execution Value (EV)†	The value after the latest execution of the test case

Note: † new concept which does not exist in initial function; * revised concept which exists in initial function.

6.2 Factor Score and Weight

The factor score indicates the level to which the factor contributes to the value of a test case. Factor weight is a percentage that indicates the contribution of the factor among the other factors in the function. The ranges of score and weight for each factor in the final value function are listed in Table 12. Internal risk, production risk, and technical risk are subjective factors, which have to be estimated by the function users in their real testing environment. The score of the factors is a relative index in whole numbers ranging from 0, indicating the lowest possible value, to *n*, indicating a higher possible value. Amount of use, function coverage, test frequency, cost of a test case, and execution value are objective factors, whose value is directly derived from the real testing environment. A higher score indicates a higher level of the factors. The cost of a test case is calculated by the cost function which is proposed by Gillenson et al. (2020). To differentiate the level of importance among the factors in the value function, factor weights can range from 0%, upwards, and can include fractional components but the sum of all the weights should equal to 1. As the value of the factor weight goes up, the impact of the factor in the

function increases. 0% indicates the factor has no contribution in the value function. 100% represents the factor fully contributes to the value function.

Table 12. Score and Weight of the Key Factors

Category		Score Range	Weight Range	
Application Level	Internal Risk (IR)	1 – n, whole number	w_1	Value range of each factor weight (0% - 100%) $w_1+w_2+w_3+w_4+w_5=100\%$
	Production Risk (PR)	1 – n, whole number	w_2	
	Technical Risk (TR)	Sum of items score	w_3	
	Amount of Use (AU)	as real value	-	
Test Case Level	Function Coverage (FC)	as real value	w_4	
	Test Frequency (TF)	as real value	-	
	Cost of a Test Case (CTC)	as real value	-	
	Execution Value (EV)	as real value	w_5	

Table 13 exhibits the range of score and weight for each technical risk item. All the items are scaled from 1 to *n* in whole number. The weight range for the technical risk items is the same as the above weight rule for the factors in the final value function.

Table 13. Score and Weight of Technical Risk Items

Category	Score Range	Weight Range
Complexity Issues (CI)	1 – n, whole number	Value range of each item weight (0% - 100%) Sum weight of all items equals to 1
Technology Issues (TI)	1 – n, whole number	
Requirements Issues (RI)	1 – n, whole number	
Personnel Issues (PI)	1 – n, whole number	
Dependency Issues (DI)	1 – n, whole number	
Previous Testing Issues (PTI)	1 – n, whole number	
Test Environment Issues (TEI)	1 – n, whole number	
Vendor Issues (VI)	1 – n, whole number	

When scoring or weighting, two general rules have to be paid attention to. The first is integrity. Users should prudently differentiate the difference among the scored/weighted factors in the function. The difference should be accurately presented by the assigned score or weight. The second is consistency. The score or weight of the same test case may vary among different users in the same context due to the diversity of subjective judgment.

In order to mitigate the potential impact of such variation, we suggest users build their own specific scoring and weighting standards.

6.3 Factor Score Normalization

The final value function aggregates different factors to obtain a final score for representing the value of a test case. Yet, this brings up an issue as each factor is measured in different units (Chatterjee & Chakraborty, 2014). For example, internal risk (no unit, but an approximate number representing the level of the risk), amount of use (numeric value), function coverage (number of features being tested), cost of a test case (currency). Although the score of the factors is restricted to the range (1-n, whole number), the score of the factors might be very discrete due to the characteristics of the factors. For instance, the score of amount of use may be 10 as the application is only used 10 times in an expected time period. Internal risk maybe scored 150 and cost of a test case is scored 3000 as some expenditure (e.g., personnel cost, testing software) occurred due to generating and executing the test case. To allow aggregation into a final value score for a test case, the score of each factor in the function has to be normalized when calculating the value. In other words, the score of each factor needs a common scale.

Normalization is widely applied in statistics as well as many other areas (e.g., management, biomedicine, psychology) for addressing the issue of measurement on different scales. One of the common methods is logarithm transformation where transform x to log base 10 of x (i.e., $x \Rightarrow \lg x$) ("Data transformation," 2020). This method is usually used for positive data which fits the scale range (i.e., 1-n) of the factors in the value function. Following this guidance, we build the normalization function as below:

$$Y_i = \lg X_i \quad (5)$$

Y_i represents the transformed value and Xi represents the factors in the value function.

As in the above discussion, the normalization function should be used before applying the value function. In other words, all the original score of the factors in the value function have to be transformed to a normalized score through the normalization function. Then the normalized score of the factors can be fed into the value function.

CHAPTER 7

GUIDE FOR APPLYING FUNCTION

Now that the value function has been developed and demonstrated, we offer helpful tips for users who may wish to apply the function's principles. In this section, we demonstrate the goal of the function, identify relevant users, and detail procedural attention matters for using the function.

7.1 Steps of Value Estimation

Given the value function as well as scoring and weighting standards, the calculation process requires a degree of orientation for proper use of the function, and for providing users a map for applying the function. There are four steps in applying the function as demonstrated in Figure 6.

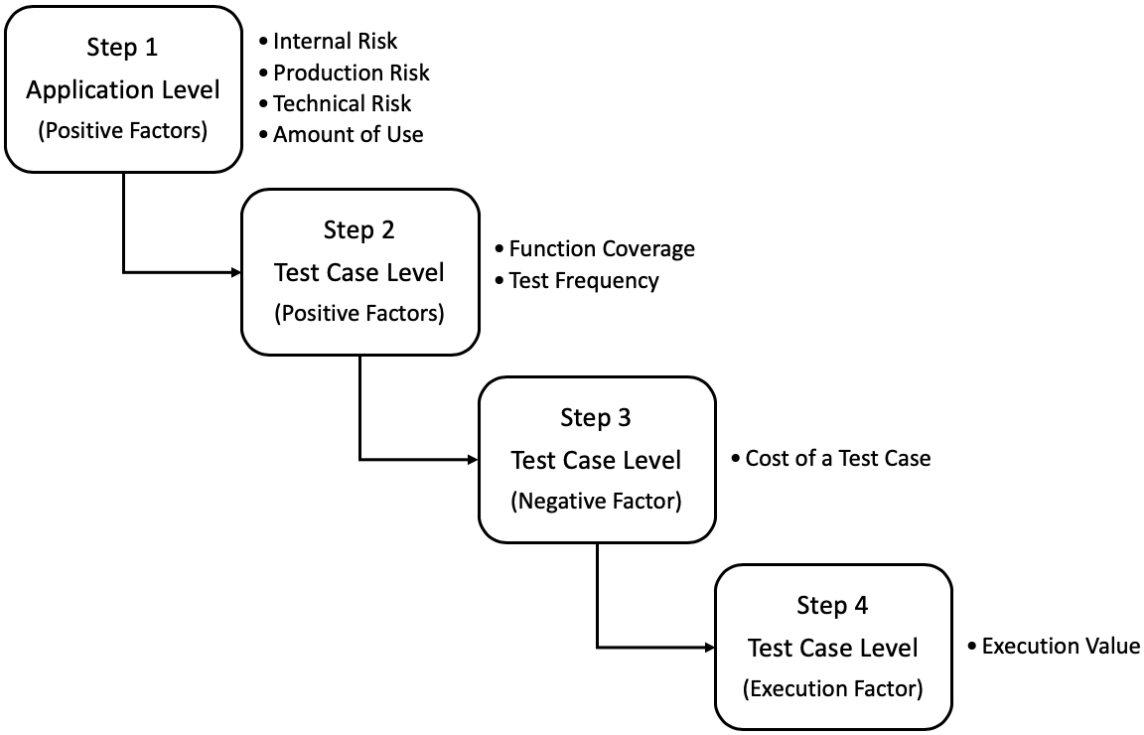


Figure 6. Steps of Applying Value Function

Step 1 is the application level procedure which processes the factors affected by the application under test. Specifically, IR, PR, TR, AU, as well as those risks' weight from w_1 to w_3 are considered. Note that the score and weight of the factors are based on the impact of the application under test, and this differs from the second step.

Step 2 is the test case level procedure that processes the positive effectiveness of the test case. Specifically, FC, TF, as well as FC weight w_4 are considered. In contrast to the first step, this step focuses specifically on the test case rather than the application under test.

Step 3 is to obtain the real cost of the test case, which is the negative factor in the test case level. We recommend setting the CTC in real cost terms which could be obtained

from financial or operational data. That would contribute to the CTC score being more reliable and consistent (Aboody & Lev, 1998).

Step 4 is to adjust the value of the test case after the test case execution. Execution value is an execution factor in the test case level. It would be effective only when the value of the test case was considerably underestimated or overestimated as the testers execute the test case. In other words, the prior three steps provide the pre-execution value of the test case, and EV in step 4 is the factor to adjust the pre-execution value to the post-execution value of the test case. EV could be zero if the real performance of the test case is close to what the test case is supposed to act.

Based on the prior four steps, the final value of the test case can be achieved by subtracting the result of in step 3 from the sum of the result of step 1, 2, and 4. Considering the wide range of test cases and testing cost and value factors across different industries, the final value may result in several potential outcomes: positive value, negative value, or zero. For the latter two results, it does not mean that a test case either has a negative value or no value. Remember that regardless of what type of outcome is, the value of the test case is just a relative value. Its value becomes meaningful only when comparing the result of one test case with that of others. As such, value calculation informs users as to which test cases are better than others, based on their relative values.

7.2 Who are the Function Users?

The purpose of the value function is to aid decisions on choosing the most valuable test cases by providing a quantitative method to discriminate between the difference of relative value rather than absolute value amongst a given set of potential test cases. To this

end, primary users might be managerial people of testing groups, ranging from middle to top level positions such as director or project manager, or testing professionals who bears responsibility for scheduling test cases execution. Other users could be departmental accountants responsible for project budgets or marketing executives responsible for the cost-effective launch of software brands.

7.3 How to Collect Value Function Data?

Although the function is reasonably easy to understand, finding the data with which to fit it is more challenging. Such data might be derived from different departments and workgroups, and in some cases it might not even be recorded by the firm. In order to reveal the value of a test case objectively and efficiently, we suggest that application-level data should be collected from marketing or business operational departments and that test-case-level data could be collected directly from testing or development groups.

For existing data, the form it might take could include budgets, accounting reports, and operational statistics. For non-existing data, estimates could be made based on corporate plans and strategic documents and from industry news reports. For example, the CTC data of a test case maybe acquired from the development or testing department, but AU data of an application might not be recorded, requiring an estimate derived from a related statistics index in the company or even from news reports or other data created by a third-party.

7.4 How to Estimate Score and Weight for Factors?

As we have indicated, nonexistent data may need to be estimated or interpolated, and this method also applies to developing the weights for factors when objective information is not available. Naturally, when data underlying key factors is estimated, this has the potential to skew unit valuations, so it is important to do so as objectively as possible. To that end, we strongly recommend that users who are developing value functions in the same company should establish, a priori, specific standards and procedures for estimating scores and weights.

7.5 How to Interpret Results?

Each application of the value function calculates a result for one specific test case, and the value is the relevant scale that is used to compare the assessed case with others to determine their relative worth in use. Therefore, when two or more test case values have been acquired from the function, the accurate interpretation of these results lies in comparisons of magnitudes of the relative value between cases. To that end, we recommend strongly against interpreting the calculated value as the real and objective value of a given test case. The value function has little meaning in consideration of only one test case. It is meant to be used as a comparative tool across cases for optimizing the selection process. There is little meaning in the process without the relative comparison process it implies. In the following section, we provide two scenarios to demonstrate using the value results within one application and across several applications.

Scenario I: Using Value Results within One Application

In this case, the value function will be used to choose the highest value test cases for the one application under consideration. Within one application, IR, PR, TR, and AU

are all the same because they are based on the same application, not on the test cases. So, they have no effect on which test cases to choose. Only the cost and other test-case-centric items, like function coverage, play a role in choosing test cases for execution. Remember that cost is subtracted from value because a higher cost makes the test case less valuable. An example of value function application in scenario I is exhibited in Table 14. There are six test cases serving for testing application A. As such, the value at application level is identical (i.e., 30), which is the result of mathematical operation among IR, PR, TR, and AU. The value differences among the test cases appear at test case level (e.g., #1 is 58, #2 is 4), resulting different total values (e.g., #1 is 88, #2 is 34), which are calculated by summing up the value at application level and that at test case level (e.g., #1 total value 88 = 30 + 58). Based on the total value from high to low, the six test cases are ranked from 1, indicating a relatively most important test case, to 6, indicating a relatively least important test case.

Table 14. Value Function Application in Scenario I

Test Case #	Application Under Test	Value at Application Level	Value at Test Case Level	Total Value	Value Rank
#1	Application A	30	58	88	1
#2	Application A	30	4	34	6
#3	Application A	30	35	65	3
#4	Application A	30	26	56	4
#5	Application A	30	17	47	5
#6	Application A	30	50	80	2

Scenario II: Using Value Results across Several Applications

In this scenario, the value function will be used primarily to allot test cases to the individual applications, in order to develop the highest quality software across the collection of applications, while trying to minimize the risks. Each test case under consideration is associated with one particular application in the application set. So, the value function result for each test case takes into account both the factors for its associated application (i.e., IR, PR, TR, and AU) and the factors of the test case itself (i.e., FC, TF, CTC, and EV). After calculating the value of each test case, the test cases can be allotted to the applications based on their relative value.

Table 15. Value Function Application in Scenario II

Test Case #	Application Under Test	Value at Application Level	Value at Test Case Level	Total Value	Value Rank
#1	Application A	30	24	54	3
#2	Application A	30	2	32	5
#3	Application B	15	2	17	6
#4	Application B	15	83	98	1
#5	Application C	27	20	47	4
#6	Application C	27	53	80	2

An example of value function application in scenario II is exhibited in Table 15, where six test cases serve for testing application A, B, and C, respectively. As such, the difference of value among the test cases appears not only at test case level (e.g., #1 is 24, #3 is 2) but also at application level (e.g., #1 is 30, #3 is 15), resulting in different total value for each case (e.g., #1 is 54, #3 is 6). The approach in calculation of value at application level, value at test case level, total value, and value rank is the same as the means in scenario I. According to the value rank, the first test case chosen will be test case

#4 for application B. The second test case chosen will be test case #6 for application C. the third test case chosen will be test case #1 for application A, and so on.

However, some adjustments may have to be made because testers probably do not want to leave even a lower priority application totally untested. Remember that IR and PR take into account the importance of the applications to the company from a business point of view. TR represents risk from a technical point of view. The concept is that the higher IR, PR, and TR are, the more valuable test cases are for that application. We have the usual assumption that testing resources are limited, leading to a limit on the number of test cases that can be used in total for several applications. So, the value function results can be used to intelligently distribute test cases across the several applications. Secondly, if it is determined that the number of test cases for a particular application is above a given threshold, then the set of test cases for that application can be reduced based on the value of each test case.

CHAPTER 8

CONCLUSION

8.1 Contributions

The potential contributions and applications of this study are threefold. First, the nature of the value in test cases explored in this study fills a notable gap in the literature, as there currently exists no specific method to determine and justify the value of test cases. Most prior research studies associated with test cases are involved in test case generation, test suite reduction, and test case prioritization. Exploring the nature of value in test cases can offer generic guidance and systematically integrate those studies.

Second, the value function reveals the essence of value for a context-specific test case and enhances an individual's decision making in allocating limited resources in

software testing. Differing from existing research focusing on test cases per se, the value function incorporates not only the direct value generated by the test cases but also the indirect value projected by the applications which the test cases test.

Last, the value function builds a foundation to create substantial parameters in test automation and artificial intelligence (AI) for software testing. Test automation can be conducted in any phase across the software testing process, which is primarily constituted by test-case design, test scripting, test execution, test evaluation, test-result reporting, and test management and other test engineering activities (Garousi & Elberzhager, 2017). To optimize test resource allocation before test case execution, the value function can be applied in the test-case design phase, generating a list of test cases to satisfy coverage criteria and engineering goals. To advance the level of test management, the value function can be used in the test management phase, which is usually conducted after test case execution for control and monitoring testing.

According to the definition coined by the Artificial Intelligence for Software Testing Association (AISTA), AI for software testing is an emerging field aimed at development of AI systems to test software, methods to test AI systems, and ultimately designing software which is capable of self-testing and self-healing (AISTA, n.d.). We believe this study has probed into the core layer of development of AI systems to test software from two perspectives (i.e., test strategy optimization, risk coverage optimization) out of the ten perspectives in reality which are delineated in the whitepaper by Philipp (2018). In test strategy optimization, the main challenge for AI systems is to find a measurement that can optimize which features to be tested in terms of the business impact derived from the features. In practice, this work still heavily relies on experts'

judgement and is performed manually. In risk coverage optimization, AI needs to find the optimal test sets which enable maximizing business risk coverage and defect detection under the given testing resource. This optimization can be achieved by mathematical algorithms.

Consdiering these challenges that practitioners face in reality, we believe our function provides a specific approach to enhance the efficiency of test automation. To imitate the human process of organizing and optimizing the set of test cases, the value function incorporates the factors associated with business, technique, and resource. The application of the function in test automation and artificial intelligence testing would considerably decrease manual work and promote the efficiency of the software testing.

8.2 Limitations and Future Research

Although we think the value function can be used as a comprehensive method for estimating the value of a test case, it does have some limitations that need to be specified herein.

First, we assume that all test cases are conducted within a waterfall software development environment. In practice, another approach widely applied in software development today is agile development, which advocates adaptive planning and evolutionary development (Lee & Xia, 2010). Compared to waterfall testing, agile testing, as part of software development in each scrum sprint, is conducted from the earliest stages to the last stage. Therefore, application development and functional test are usually completed in the same phase. In this situation, application level factors, such as estimating the risks that arise due to application defects, cannot be taken into consideration in the

value function. Moreover, agile development is driven by testing where providing testing results in a timely manner for the agile development group is the core feature of test cases. In this dynamic environment, it is unlikely to have the situation where testers or developers select the most valuable test cases among several choices at the same time. In a future research project, we recommend extending our value function to enable it to adapt to the agile testing environment. Another direction for future research is applying the value function to all types of test cases because only functional test cases are considered in this study. In practice other types of test cases, such as test cases for security testing, also play a significant role in detecting defects in applications and consume considerable test resources. To optimize the allocation of the entire testing resource, all types of test cases need to be considered and estimated by an appropriate standard.

Second, the interview data was collected from one organization, which may limit the generalizability of the value function. Although the company we chose in the study has a large number of test professionals around the world, it is still difficult to represent all of the available software testing contexts. The remedy for enhancing the value function in future research is to obtain interview data from multiple companies in different industrial sectors.

Last, the value function is a conceptual model which heavily relies on a subjective estimation technique, especially for the "cold start," the early stage of utilizing the function. A subjective estimation technique is advocated for the situations, where there is no initial data or weighting information is difficult to obtain. The results can only be as good as the dedication to standards in the subjectivity in estimation that is employed when real data is not easily available. As more data and experience are gained from using the

function, the results of value estimation should become more reliable and more accurate. Therefore, applying the value function in real software test environments and evaluating its effect is another future research direction.

REFFERENCES

Aboody, D., & Lev, B. (1998). The Value Relevance of Intangibles: The Case of Software Capitalization. *Journal of Accounting Research*, 36, 161-191.

Albrecht, A. J., & Gaffney, J. E. (1983). Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, (6), 639-648.

AISTA (n.d.). *Artificial Intelligence for Software Testing*. https://www.aitesting.org/

Baker, C. (1957). Review of D.D. McCracken's "Digital Computer Programming". *Mathematical Tables and Other Aids to Computation*, 11(60), 298-305.

Beck, K. (1999). Embracing Change with Extreme Programming. *Computer*, 32(10), 70-77.

Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., & Grünbacher, P. (Eds.). (2006). *Value-Based Software Engineering*. Springer Science & Business Media.

Boehm, B. W. (1981). *Software Engineering Economics*. Upper Saddle River, NJ, USA: Prentice Hall.

Boehm, B. W. (2006). *Value-Based Software Engineering: Overview and Agenda.* In Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., & Grünbacher, P. (Eds.), *Value-Based Software Engineering* (pp. 3-14). Springer.

Boehm, B., & Huang, L. G. (2003). Value-Based Software Engineering: A Case Study. *Computer*, 36(3), 33-41.

Boehm, B. W., & Papaccio, P. N. (1988). Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10), 1462-1477.

Burnstein, I. (2006). *Practical Software Testing: A Process-Oriented Approach*. Springer Science & Business Media.

Cambridge Dictionary. (n.d.). Value. In *Cambridge Dictionary*, Retrieved April 4, 2020, from https://dictionary.cambridge.org/us/dictionary/english/value#dataset-cald4.

Charette, R. N. (2005). Why Software Fails. *IEEE Spectrum*, 42(9), 42-49.

Chatterjee, P., & Chakraborty, S. (2014). Investigating the Effect of Normalization Norms in Flexible Manufacturing Sytem Selection Using Multi-Criteria Decision-Making Methods. *Journal of Engineering Science & Technology Review*, 7(3), 141-150.

Cohen, M. B., Gibbons, P. B., Mugridge, W. B., & Colbourn, C. J. (2003, May). Constructing Test Suites for Interaction Testing. *In Proceedings of the 25th International Conference on Software Engineering* (pp. 38-48). IEEE Computer Society.

Cook, J. E., Votta, L. G., & Wolf, A. L. (1998). Cost-Effective Analysis of In-Place Software Processes. *IEEE Transactions on Software Engineering*, 24(8), 650-663.

Cooper, R., & Kaplan, R. S. (1988). Measure Costs Right: Make the Right Decisions. *Harvard Business Review*, 66(5), 96-103.

Cresswell, A. M. (2004). *Return on Investment in Information Technology: A Guide for Managers*. Center for Technology in Government, University at Albany, SUNY. http://www.ctg.albany.edu/media/pubs/pdfs/roi.pdf

Data transformation (statistics). (2020, January 12). In *Wikipedia*. https://en.wikipedia.org/wiki/Data_transformation_(statistics)

Deutsch, M. S. (1981). Software Project Verification and Validation. *Compurer*, 14(4), 54-70.

Dreger, J. B. (1989). *Function Point Analysis*. Prentice-Hall.

Felderer, M., & Ramler, R. (2014). Integrating Risk-Based Testing in Industrial Test Processes. *Software Quality Journal*, 22(3), 543-575.

Garousi, V., & Elberzhager, F. (2017). Test Automation: Not Just for Test Execution. *IEEE Software*, 34(2), 90-96.

Gefen, D., Wyss, S., & Lichtenstein, Y. (2008). Business Familiarity as Risk Mitigation in Software Development Outsourcing Contracts. *MIS Quarterly*, 32(3), 531-551.

Gelperin, D., & Hetzel, B. (1988). The Growth of Software Testing. *Communications of the ACM*, 31(6), 687-695.

Gillenson, M. L., Stafford, T. F., Zhang, X. & Shi, Y. (2020). Use of Qualitative Research to Generate a Function for Finding the Unit Cost of Software Test Cases. *Journal of Database Management*, 31(2), 42-63.

Goodenough, J. B., & Gerhart, S. L. (1975). Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, (2), 156-173.

Griffin, D. (2019). *Types of Business Risk*. Chron. http://smallbusiness.chron.com/types-business-risk-99.html

Hass, A. M. (2014). *Guide to Advanced Software Testing* (2nd ed.). Norwood, MA: Artech House.

History of Microsoft Office. (2020, March 3). In *Wikipedia*. https://en.wikipedia.org/wiki/History_of_Microsoft_Office

Hoodat, H., & Rashidi, H. (2009). Classification and Analysis of Risks in Software Engineering. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 3(8), 2044-2050.

IEEE. (1986). *1012-1986 - IEEE Standard for Software Verification and Validation Plans*. https://standards.ieee.org/standard/1012-1986.html

ISO/IEC/IEEE. (2017). *24765:2017 Systems and Software Engineering — Vocabulary*. https://www.iso.org/standard/71952.html

ISO/IEC/IEEE. (2011). *42010:2011 Systems and Software Engineering — Architecture Description*. https://www.iso.org/standard/50508.html

Iversen, J. H., Mathiassen, L., & Nielsen, P. A. (2004). Managing Risk in Software Process Improvement: An Action Research Approach. *MIS Quarterly*, 28(3), 395-433.

Jalote, P., & Vishal, B. (2003). Optimal Resource Allocation for the Quality Control Process. In *Proceedings of the 14th International Symposium on Software Reliability Engineering, Denver, Colo, USA, November 2003*.

Jorgensen, M. (2004). A Review of Studies on Expert Estimation of Software Development Effort. *Journal of Systems and Software*, 70(1-2), 37-60.

Jorgensen, M. (2005). Practical Guidelines for Expert-Judgment-Based Software Effort Estimation. *IEEE Software*, 22(3), 57-63.

Jorgensen, P. C. (2018). *Software Testing: A Craftsman's Approach* (4th ed.). CRC Press.

Juristo, N., Moreno, A. M., & Strigel, W. (2006). Guest Editors' Introduction: Software Testing Practices in Industry. *IEEE Software*, 23(4), 19-21.

Kazman, R., Asundi, J., & Klein, M. (2001). Quantifying the Costs and Benefits of Architectural Decisions. In *Proceedings of the 23rd International Conference on Software Engineering* (pp. 297-306). Washington, DC, USA: IEEE Computer Society.

Kohli, R., & Grover, V. (2008). Business Value of IT: An Essay on Expanding Research Directions to Keep up with the Times. *Journal of the Association for Information Systems*, 9(2), 23-39.

Lee, Y. W., Strong, D. M., Kahn, B. K., & Wang, R. Y. (2002). AIMQ: A Methodology for Information Quality Assessment. *Information & Management*, 40(2), 133-146.

Lee, G., & Xia, W. (2010). Toward agile: an integrated analysis of quantitative and qualitative field data on software development agility. *MIS quarterly*, 34(1), 87-114.

Lin, Y., Chou, C., Lai, Y., Huang, T., Chung, S., Hung, J., & Lin., F. (2012). Test Coverage Optimization for Large Code Problems. *Journal of Systems and Software*, 85(1), 16-27.

Marshall, B., Cardon, P., Poddar, A., & Fontenot, R. (2013). Does Sample Size Matter in Qualitative Research?: A Review of Qualitative Interviews in IS research. *Journal of Computer Information Systems*, 54(1), 11-22.

Mathur, A. P. (2013). *Foundations of Software Testing* (2nd ed.). Pearson Education India.

Melville, N., Kraemer, K., & Gurbaxani, V. (2004). Information Technology and Organizational Performance: An Integrative Model of IT Business Value. *MIS Quarterly*, 28(2), 283-322.

Miller, E., & Howden, W. E. (Eds.). (1981). *Tutorial: Software Testing and Validation Techniques*. IEEE Computer Society Press. New York.

Molokken, K., & Jorgensen, M. (2003). A Review of Software Surveys on Software Effort Estimation. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering* (pp. 223-230). IEEE.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing*. John Wiley & Sons.

Myers, M. D. (1997). Qualitative Research in Information Systems. *MIS Quarterly*, 21(2), 241-242.

Myers, M. D. (1999). Investigating Information Systems with Ethnographic Research. *Communications of the Association for Information Systems*, 2(23), 1-20.

Neumann, A. J. (Ed). (1982). *NBS FIPS Software Documentation*. Institute for Computer Sciences and Technology, National Bureau of Standards. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-94.pdf

New World Encyclopedia. (2016). Value, Philosophical Theories of. In *New World Encyclopedia*. Retrieved April 4, 2020, from

http://www.newworldencyclopedia.org/p/index.php?title=Value,_Philosophical_theories_of&oldid=993311

Perry, W. E. (2007). *Effective Methods for Software Testing: Includes Complete Guidelines, Checklists, and Templates*. John Wiley & Sons.

Philipp, I. (2018). *AI in Software Testing: A Reality Check*. Tricentis. https://www.tricentis.com/resources/ai-in-software-testing-reality-check/

Phillips, J. J. (Ed.). (1994). *In Action: Measuring Return on Investment*. American Society for Training and Development.

Porter, M. E. (1985). *Competitive Advantage: Creating and Sustaining Superior Performance*. New York: FreePress.

Porter, M. E., & Millar, V. E. (1985). How Information Gives You Competitive Advantage. *Harvard Business Review*, 63(4), 149-160.

Ramler, R., Biffl, S., & Grünbacher, P. (2006). Value-Based Management of Software Testing. In Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., & Grünbacher, P. (Eds.), *Value-Based Software Engineering* (pp. 225-244). Springer.

Rapoport, R. N. (1970). Three Dilemmas in Action Research: With Special Reference to the Tavistock Experience. *Human Relations*, 23(6), 499-513.

Saldaña, J. (2015). *The Coding Manual for Qualitative Researchers*. Sage.

Schwaber, K. (1997). SCRUM Development Process. In J. Sutherland, C. Casanave, J. Miller, P. Patel, & G. Hollowell (Eds.), *Business Object Design and Implementation* (pp. 117-134). Springer.

Talby, D., Keren, A., Hazzan, O., & Dubinsky, Y. (2006). Agile Software Testing in a Large-Scale Project. *IEEE Software*, 23(4), 30-37.

Tassey, G. (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing*. National Institute of Standards and Technology. https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf

Tricentis. (n.d.). *Software Fail Watch: 5th Edition*. https://www.tricentis.com/resources/software-fail-watch-5th-edition/

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, 49, 433-460. http://cogprints.org/499/1/turing.html

Van Solingen, R. (2004). Measuring the ROI of Software Process Improvement. *IEEE Software*, 21(3), 32-38.

Wallace, D. R., & Fujii, R. U. (1989). Software Verification and Validation: An Overview. *IEEE Software*, 6(3), 10-17.

Whittaker, J. A. (2000). What is Software Testing? And Why is It So Hard? *IEEE Software*, 17(1), 70-79.

Wohlin, C., & Aurum, A. (2006). Criteria for Selecting Software Requirements to Create Product Value: An Industrial Empirical Study. In Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., & Grünbacher, P. (Eds.), *Value-Based Software Engineering* (pp. 179-200). Springer.

Yiftachel, P., Hadar, I., Peled, D., Farchi, E., & Goldwasser, D. (2011). The Study of Resource Allocation among Software Development Phases: An Economics-Based Approach. *Advances in Software Engineering*, 2011, 1-21.

Yin, R. K. (2017). *Case Study Research and Applications: Design and Methods*. Sage Publications.

APPENDIX A

An Example of Functional Test Cases

Functional test case is created for checking the functionality of a program. In a test scenario, one or several test cases are initiated by specifying input values, expected results, and other relevant elements which support executing test, such as test pre-condition, test steps, etc. Through comparing the actual result and expected result of the test cases, defects are found when two results are not reconciled. This situation is also called test fail.

In Table 16, we provide an example of functional test cases for checking login functionality on FedEx company's homepage. Given the scenario, four test cases with their IDs (T001, T002, T003, T004) are created for verifying the login functionality in four possible situations. All the test cases start from the same pre-condition where the homepage is on log off status as shown in Figure 7. Next, the four test cases are executed separately in the same test steps: 1. go the site (http://www.fedex.com); 2. click "Sign Up or Log In" button; 3. enter user ID; 4. enter password; and 5. click "LOGIN" button. Note that four different combinations of user ID and password are given to the four test cases. In the meanwhile, four expected results are also established for the test cases based on their designated user ID and password (see Table 16).

Table 16. An Example of Test Cases

Test Scenario	Test Case ID	Pre-condition	Test Steps	Test Data*	Expected Results	Actual Results	Pass/ Fail
Check login function on the homepage	T001	1. homepage is on log off status.	1. Go the site http://www.fedex.com 2. Click "Sign Up or Log In" button 3. Enter user ID 4. Enter password 5. Click "LOGIN" button	User ID: ABC (valid) Password: 12345678 (valid)	1. User's name (ABC) should be displayed next to "log off" button on the right top corner.	As expected	Pass

	T002			User ID: AAA (invalid) Password: 12345678 (valid)	1. User should not login. 2. No any name should be displayed on the right top corner. 3. Display the information “Login incorrect...”	As expected (see Figure 8 right image)	Pass
	T003			User ID: ABC (valid) Password: 000000 (invalid)		As expected (see Figure 8 left image)	Pass
	T004			User ID: AAA (invalid) Password: 000000 (invalid)		As expected (see Figure 8 right image)	Pass

* The user IDs and passwords are not real data and only used for the example demonstration.

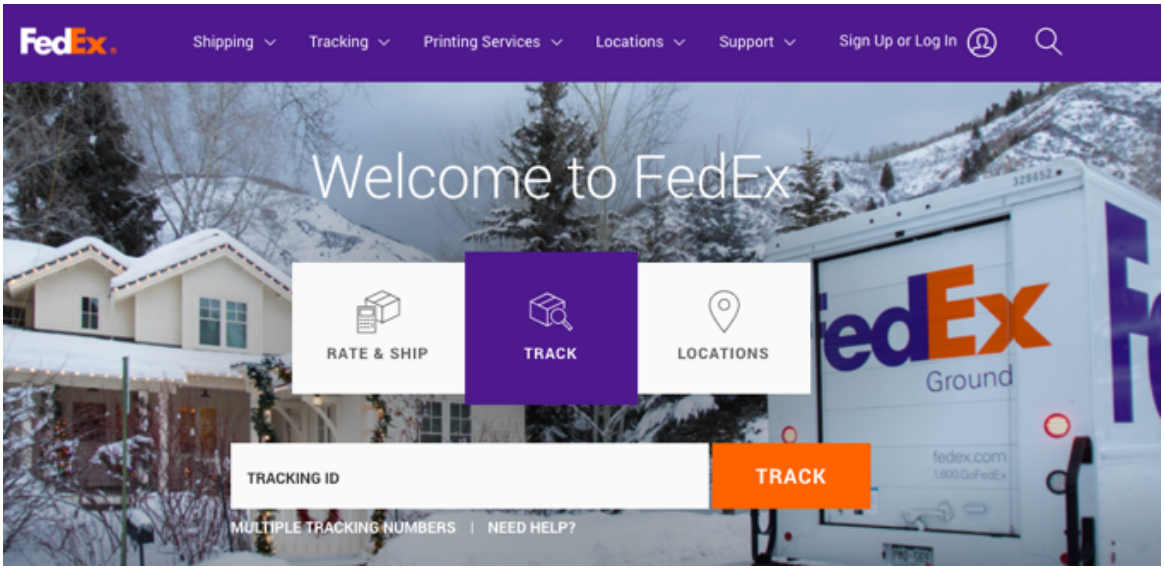


Figure 7. Pre-condition of the Test Cases

Specifically, T001 is designated with a valid user ID “ABC” and its valid password “12345678”. Its expected result is that user’s name “ABC” should be displayed next to “log off” button on the right top corner of the homepage. In contrast, T002, T003, and T004 are designated with either an invalid user ID or an invalid password or with both

(T002: invalid user ID “AAA” and valid password “12345678”; T003: valid user ID “ABC” and invalid password “000000”; T004: invalid user ID “AAA” and invalid password “000000”). Since the login functionality is the portal that prevents anyone from accessing FedEx’s systems by using any invalid user ID or password, the expected result of T002, T003, and T004 should appear as follows: 1. user should not login; 2. no any name should be displayed on the right top corner of the homepage; and 3. display the information “Login incorrect...” as shown in Figure 8.

For each test case, the actual result after executing the test case has to be compared with the expected result. The test is passed if the two results are exactly the same. Otherwise, the test is failed while extra procedure is required for investigating the root of the failure. Figure 8 shows T002 and T004’s actual results in right image and T003’s actual result in left image. Those actual results are the same as their expected results. Therefore, those tests are passed.

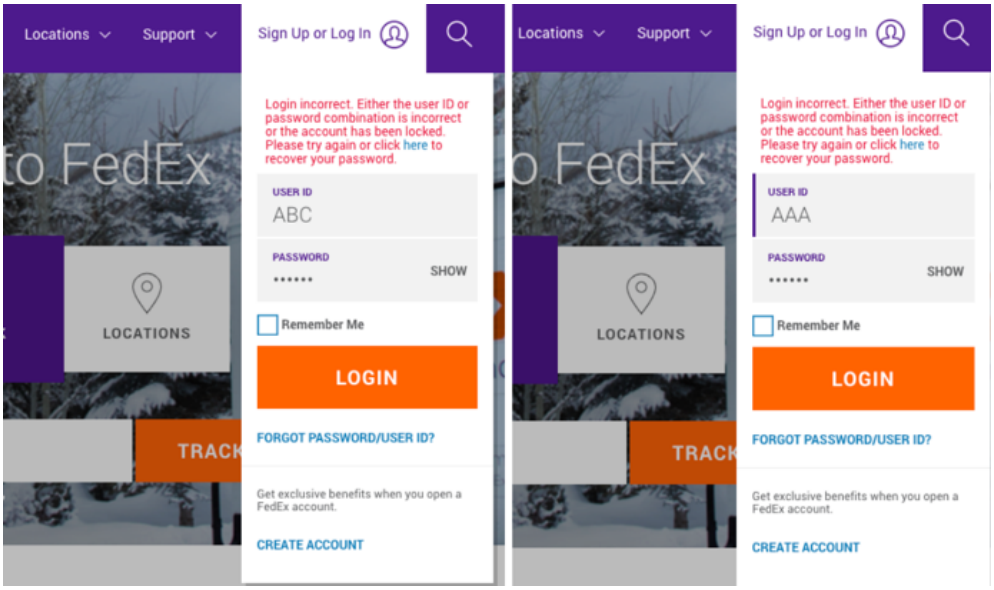


Figure 8. Test Result of the Test Cases

APPENDIX B

Interview Instrument

This interview is related to evaluating value of a functional test case. Each participant is assumed that they have a bunch of test cases that can be applied, but to implement all of them is not practical. In this situation, participant should make some choices from those tests based on their testing experience and justification. Please answer the following questions and give your reason (see Table 17).

Table 17. Interview Instrument

Interview #: ( ) Company: ( ) Interviewee: ( ) Interview Date: ( )	
Interview Questions	Note
1. Do you agree or disagree with the initial function of unit value of a test case if you consider Unit Value is the key factor to impact your choice? Why?	
2. If you agree with the initial function of unit value of a test case, should Business & Operational Risk, Technical Risk and Unit Cost be expanded in the function, as opposed to having a separate table for its components? Why?	
3. Do you agree or disagree with the components of Business & Operational Risk and Technical Risk? Give your opinion?	
4. Do you think Weight, standard scale setting for each factors and Amount of Use can establish a relatively clear and practical approach to assess the unit value of a test case? What's your opinion?	
5. Do you have any other comments on this function?	

APPENDIX C

Code Book v1

Table 18. Code Book v1

Code	Definition	Interview Transcript
Simplicity	To simplify the function or to expand the function in detail.	1(1) To keep the function simple, expand out TR and UC in accompanying tables. 7(6) Do not promote any line items in the table into the main function. It would get too complicated. 8(6) Keeping function simple would help people readily comprehend the meaning of the function. Do not promote table rows into the function. 10(8) Leaving the function with a limited number of factors while providing the table below it with further details is good. 13(5) Leave the mail function as it is. Don't expand any of the factors in the main function. 13(6) Leave the function as multiple factors; don't try to combine them. 14(7) Do not expand TR and UC in the function. 16(6) It is not necessary to expand the UC factor in the main function. 21(8) Too much detail in the function could be confusing. 25(8) Keeping the equation simple without promoting the TR line items to the equation is a good way to present the unit value of a test case.
Priority	A factor has relative higher or lower level of importance against another factor.	1(2) Risk is not equal to priority. The priority should be considered as a separate factor in the function. 2(1) A priority factor should be added in the function. Priority might be rated as high, medium, and low. 4(1) The priority of a test case is influenced by how critical the application feature is that the software is implementing, such as handling customer complaints, dealing with cutting edge technologies being used by competitors, etc. 5(1) The priority of a test case should be a separate factor. But the relationship between revenue generation and priority is very limited, since it is hard for a testing group to figure out the amount of revenue generated by a feature or application. 8(1) Priority always accompanies risk, especially in internal risk and external risk. It is hard to list priority separately. The standards

		of priority vary from one case to another. Sometimes satisfying customers, which is an element of external risk, is prioritized. In some other situations, release time, which is an element of internal risk, is considered the most critical element. 8(2) Every risk is a matter or measure of priority. They are two sides of the same coin. 9(1) Risk significantly differs from priority. Priority is an aspect of risk. Priority varies in different contexts. Priority might focus on the number of customers affected or what would happen to the brand if a problem hit the media. 9(2) Priority is a piece of both Internal Risk (IR) and External Risk (ER). 9(4) Death is a top priority. Safety is a top priority. 10(2) The priority of a piece of software falls if there are work-arounds that render the software unnecessary. 11(1) The priority of a test case is not the same as the unit value of a test case, but they are closely related to each other. 12(1) Priority can be based on different reasons. How important is the software to the customer, to the business, or to marketing efforts? 14(4) Priority is a part of risk; high priority leads to high risk. For example, the media attention of the software would raise the ER of the test case.
Amount of Use	The amount of use of the application.	1(3) The amount of use is a very critical factor. Multiplying external risk by it might be insufficient to elaborate its important role. Multiplying the entire set of risk factors is an option for this point. Different people may have different views on the importance of amount of use. 2(5) Amount of use, AU, is an ideal concept for estimating the importance of a test case. It may be easier to estimate AU for a customer-facing application than for an internal application. 2(7) Uncertain about whether AU should be expanded to multiply more risk factors or not. 2(8) The estimate of AU varies depending on whether the software is a new application or a revision of an existing application. 3(4) The amount of use is an expected and subjective number, not an accurate number in practice. Multiplying the external risk by the amount of use makes more sense than multiplying the entire risk by the amount of use. 4(2) It is better to multiply the entire risk by the amount of use (AU) rather only multiplying the external risk by the amount of use.

		5(3) Multiplying only ER by AU is appropriate in the function. 6(4) It makes sense to use AU for calibrating the UV of a test case. But there are several options (listed below) for AU to combine with other factors. The AU could be considered as the probability of software failure. (IR+ER)*TR*AU/UC IR+(ER*TR*AU)-UC (IR+ER+TR)*AU-UC 7(3) It is appropriate to include AU as it currently appears in the function. It applies to External Risk (ER), not to the other risk factors. 8(8) AU is particularly relevant to External Risk (ER.) 8(11) External Risk (ER) is the only factor that should be multiplied by Amount of Use (AU). 10(1b) Customer satisfaction, amount of use, etc. In a word, all of those perspectives are subject to or related to revenue generation, including not overcharging or undercharging customers. 10(5) ER multiplied by AU is preferable, since AU is much more directly correlated with ER than with the remaining factors. If the software fails, you are going to lose revenue or customers. 11(4) Multiplying ER by AU is better than multiplying the other factors by AU. 12(3) AU Multiplying only ER, rather than the other factors, by AU, is correct. 13(4) Leans towards multiplying all three risk factors by Amount of Use (AU). 14(1) Agree with the AU multiply by ER. If the software has higher customer rate, it implies the software would be used much more frequently and the highest risk is if it affects the most customers. 15(1) AU is related to some aspects of ER rather than all of the elements of ER. For instance, relevant regulations are part of ER but are not affected by AU, while other aspects of ER are affected by AU. 15(6) Multiply only ER by AU, not the other risk factors. 16(1) Multiply all risks by AU. 17(1) The AU should multiply all three risk factors: IR, ER, and TR. 18(8) AU influences the value in revenue potential. This is separate from IR and ER. 18(9) The insight for the function: (IR+ER) [AU-(UC+TR)]. Note that this implies that AU multiplies all risk factors.

		19(1) AU should multiply all risk factors: IR, ER, and TR. 19(2) AU may not always be a good indicator of the value of a test case. One-time use of an application could be critical and present a very high risk if the software fails. For example, signing-up a new customer could mean the loss of the customer if the application fails. This also implies that AU and revenue generation may not directly correlate. 20(2) AU should multiply all of IR, ER, and TR. 20(8) AU could mean how many new customers need to be signed-up, not just how many times the customers use the application. 22(1) AU should only multiple ER. 22(2) AU of the critical path is crucial. You have to be careful about multiplying ER by AU because not all features of the software might be used all of the time. 23(5) The IR of a piece of software increases if its use goes across multiple divisions within a company. Therefore, test cases that test this software are more valuable. 23(6) External risk increases if the software is intended to be used in many countries on a global basis. Therefore, test cases that test this software are more valuable. 23(10) AU should be associated with "failure costs" in ER. AU should also multiply TR. 24(7) Different parts of the code may be important for different reasons. The code for particular exception conditions may be very important even if infrequently used. Therefore AU is only one of the influential factors representing the critical level of the software. In some cases, software may have very low AU but still have a very high critical level which cannot be ignored. 26(1) AU should multiply both IR and ER. 27(1) AU multiplying just ER or multiplying all risk factors could go either way.
Special Case	A special case of using the function.	1(4) The special case, high risk but low amount of use, should be demonstrated.
External Risk	Risks arising from the events taking place outside of the organization.	1(5) Look into risk in other areas, such as mergers and acquisitions in the financial field. 6(2b) Regulations, and unique feature with extremely competitive strength in the market. It is appropriate to separate priority from the risks. 6(3) External risk factors influence internal risk factors. 7(1) When considering the priority of test cases, risks such as crucial impacts of failure in production and relevant regulations, as

		well as other risk items listed but not limited in the table, always influence the priority of a test case. Therefore, priority should not necessarily be listed as a new factor in the function. Priority should be listed as another row in Internal Risk (IR). 7(7) In External Risk (ER), split out crucial versus non-crucial impacts. For example, a failure that impacts revenue is one kind of problem while a usability issue is another kind of problem. 10(3) The regulation issue listed in ER is not parallel with the legislation issue. In some cases, products may not violate applicable laws but could be in violation of conventional regulations. 11(2a) Business impact could impact the priority of the software. This could be an additional line item in TR. 11(3) ER is based on the probability of the software failing in production. 15(8) An issue of ER is impact to the brand. 18(6) The value of a test case depends on the potential loss of customers if the software fails in production. 23(8) Test cases that test software that could affect customer satisfaction are more valuable. 24(8) ER should be changed to an overall production risk (PR) to take into account both external facing and internal facing applications. 25(1) The ER could be affected by the impact from social media that may influence the potential customers' judgement. 25(2) ER can be considered as a production risk because the application may be either external facing or internal facing. 27(2) Production risk as a general issue is more realistic than what we had been considering as failure in production of an external application.
Weight	The weight of factors in the function.	1(6) Write a description of how to use the weights in the function. 14(5) Weights should be different for each factor because each item has different effect and risk in different context. 21(7) Weights can go to zero if a factor is not important.
Internal Risk	Risks arising from events taking place within the organization.	2(2) They currently consider priority, but do not consider executive pressure, which is another reason for splitting out priority as a separate factor. 2(3) Executive pressure may be about quality or speed of development, or both. 6(2a) An additional factor is urgency, such as the deadline for release.

		8(3) Executive pressure in Internal Risk (IR) can go two ways. “Do it well,” in which case the unit value of a test case should go up; “Do it fast,” in which case the unit value of a test case should go down. 8(4) If a test case encounters resource limitations such as budget, time, or personnel, the value of the test case may appear to decrease even though the effectiveness of the test case is significant. In other words, it’s not the test case’s fault if the project has run out of time or other resources for it. For this research project, the assumption has to be that if a test case is being considered, there is enough time for it. 8(10) Change the rows in the table for Internal Risk (IR) to time, quality, and cost. 10(4) Both high quality and time pressure apply to IR. 13(1) Need to consider business criticality/mission criticality in Internal Risk (IR). “Tier 1” mission critical system. 13(2) Speed to market and agility are Internal Risk (IR) factors. 15(5) Regarding IR, is the test case important “to someone who matters?” 15(9) IR can change over time, especially in agile development as requirements change. 16(3) Executive pressure in IR could be dependent on the area of the company the application is being developed for. 18(7) Risk factors are assigned at different levels of the company. For example, IR is assigned at the executive level. Testers on the front lines, normally have to follow the high level managers’ directions regarding risk factors. 19(7) IR includes time pressure, resource pressure, and opportunity cost. 20(6) The time dimension is a critical factor that should be added into the function. The unit value of a test case is subject to release time pressure in IR and competitive pressure in ER. There is a tradeoff: Time pressure is a zero-sum game. If there is time pressure it may be more important to have defect-free software but it requires more time to test.
Unit Cost	Comments regarding the cost of a test case.	2(4) Unit cost is always evaluated in terms of dollars, especially in those departments concerned with preparing budgets. Those departments usually can estimate a relatively accurate estimate of the unit cost based on historical records. 15(4) Breaking down UC into two categories: one time costs (preparation costs, creation costs) and multiple times costs (run costs, failure cost), would make clear sense.

Technical Risk	Risks arising from technical perspectives regarding the application.	2(6) The function also should be considered from vendor group perspective rather than only from testing group perspective. "Vendor issues" should be a line item in TR. 3(7) Add History of the SUT as an additional Technical Risk (TR) line item in the table. 3(8) In the Technical Risk (TR) table, replace "dependency issues" with the degree to which this piece of software affects or interacts with other pieces of software. 4(4) Maintaining the line items of Technical Risk (TR) in a table rather than putting each of them separately in the unit value function is a good idea. 7(2) Since test cases always encounter various kinds of Technical Risk (TR) in the software under test, each line item of TR listed in the table should be weighted separately and then combined to form the total TR. Without this there is a degree of inconsistency. This may not be necessary for IR and ER. 9(5) The time factor, which often influences a decision maker's judgment, should be taken into account in the function as a significant factor. A proposed test case which, based on history, is projected to run in a shorter amount of time is more valuable. 9(6) An automated test case is more valuable than a manual test case. 10(6) All the items listed in TR help people from diverse perspectives evaluate the confidence of the testing staff for completing a test case well. Different testing groups in different situations, however, perceive different TR line items to be more or less important. Thus, the TR line items should be weighted separately to allow for the needed diversity. 11(2b) level of change of the application are two dimensions that could impact the priority of the software. This could be an additional line item in TR. 11(7) Another line item in TR should be the complexity of database interfaces and issues of the application being in the cloud. 12(4) In his situation, complexity and dependency of the software are more critical than the other items of TR. However, the critical level of the items could vary in different situations, so having TR line items with separate weights is a good solution. 13(7) Difference in the value of a test case based on whether it's new or if we have experience with it. 16(4) The newness of the testing technology, either to the industry or to the company, is a factor in TR.

		16(5) Cooperation in different departments or groups is common and is a factor in TR. 17(4) It is better to break down the TR into several subcategories, each of which has its own weight. 19(8) We need more detail for each TR factor, including individual weights. 21(11) A new line for TR is the complexity of the test data. 21(12) The previous testing line in TR includes "brittle code." 21(13) TR includes badly designed test cases. 21(14) A poor test environment doesn't make a test case more or less valuable. 21(15) Does the "test environment issues" line in TR belong there? 22(6) The TR line items should have individual weights. 22(7) The TR line items should be promoted into the main function. 22(8) Another TR line item should be architectural complexity, e.g. asynchronous versus synchronous web service calls. 23(9) The line items in TR should be weighted separately. 25(7) TR is also influenced by the project, which can be called project risk. Project risk is normally caused by limited resources, such as having a fixed date by which the project must be completed. 26(4) Another line item for TR is if the code comes in late to the testers. 27(3) Lacking staff or other resources such as servers because of a delay in acquiring ordered hardware are TR factors. 27(5) Regime change, i.e. changing from full-time employees to contractors is a TR factor. 27(9) The individual line items of TR should be weighted individually.
Unit Value	General comments regarding the component and structure of the value function.	3(1) The unit value may be positive or negative, it depends on whether the risk value is greater than the cost. 5(2) The Unit Value of a test case is projected before application execution rather than after the process. 5(4) There is no difference in the value of a test case whether it tests a small piece of software or a large piece of software.

		5(8) Keep the unit value function simple while providing a table with details below it. 5(9) Future research: What is the value of continuing to have a test case in a regression suite? 7(4) A test case that goes into a regression suite is more valuable than one that does not. 7(5) A test case that tests a series of applications is more valuable than one that does not. 8(5) The meaning of unit value is to explain why one test case should be implemented versus another. 9(3) Some items of Internal Risk (IR) and External Risk (ER) should be clarified, such as "crucial impacts of failure in production". 9(9) The definition of unit value should be clarified. In different contexts, it may be comprehended as customer satisfaction-oriented, or revenue generation-oriented, or margin increase-oriented, or other related perspectives. The value of a test case depends on the context in which it is used. 9(10) Comparing the value of different groups of test cases is future research. 9(11) We have to define what we mean by "value." 13(3) If this is a good test case, the testing process will be better. 13(8) Quality of a test case is important and is based on the number of defects found by it. 13(9) A test case in a regression suite is more valuable than one that is not. 14(2) Adding a test case to the regression suite is not a necessary condition for evaluating the value of the test case. A special test case that is specifically targeted for a reason and used once can be just as important as a test case that goes into a regression suite. 14(3) A test case that finds no defect is just as valuable as one that finds defects. This has no effect on the value of a test case. 14(6) In practice, choosing a test case among several choices depends on good guess or experience the test group has. The value function is a good tool for testing people in selecting an appropriate test case in terms of the value. 14(8) Using this value function will help prioritize the work to make the products better.

		14(9) Recursive use of the value function – do you find greater value after you start running the test case because, for example, it finds a lot of defects. 15(2) The value of a test case may change as you use it in testing. There is a scenario that is pervasive in practice. A test case that was considered to be low value in the initial stage may increase in value due to more defects being detected after the test case execution. 15(3) The value of a test case may increase as you use it in testing as you realize that the code it is testing is more complex than originally thought. 15(7) The value of a test case changes over time and so value should be considered to be in a feedback loop. 16(7) Normally, they would add a test case into the regression suite unless it's too complex to run. This is not a matter of the test case's value. 16(8) The value of a test case increases if it detects defects in risky code. 16(9) "After the fact" increases in test case value can occur if the test case finds defects. This could cause you to decide to add it to the regression suite. 17(3) The value of a test case increases somewhat if it is added to a regression suite. 18(1) Whether the test case is eligible to be added into a regression suite cannot significantly affect the value of the test case because all test cases are added to a regression suite. 19(3) If a test case is added into a regression suite, it indicates that the test case has higher value than test cases that are not added to a regression suite. 19(5) The value function should be considered as a dynamic function rather than a static function because the value of a test case may change after the test case is executed. 19(6) A test case is more valuable if it is used in end-to-end testing. 19(9) The entire value model should be dynamic because everything can change, "in a heartbeat." 20(3) We are not comparing adding a test case at the unit level to another level. 20(5) The value of a test case is determined up front and a high-value test case is added into the regression suite. Value is not determined by the decision of whether or not to add it to the regression suite.

		20(7) Risk values should not be on a linear scale but should be on an exponential, modified Fibonacci scale. Doing this may eliminate or reduce the need for weights. 21(4) Adding a test case into a regression suite could be the standard to evaluate the value of a test case. 21(5) Especially for a new system where you're not sure about the critical path, the unit value of a test case function could be dynamic. 21(6) Risk values should be on an exponential scale. 21(9) A test case that is targeted to a part of an application is just as valuable as a test case that goes into a regression suite. 21(10) Another use of the unit value of a test case function is to reevaluate the test cases in an existing regression suite. 22(3) The value of a test case should be a factor of producing revenue or reducing cost. 22(4) The value of a test case should be based on the function points (i.e. requirements) instead of the amount of code or of specific parts of the code covered. 22(5) The most valuable test cases are the ones that go into the regression suite. 22(9) Risk values should be on an exponential scale. 23(2) A test case is more valuable if it tests an application in such a way that it makes sure that applications that are communicate with it are not adversely affected. 23(3) A test case is more valuable if it covers multiple countries that an application is intended to be used in. 23(4) The value of a test case is based on the business value of the software under test. 24(1) ScaledAgileFramework.com (SAFe) orders the development of software as the "weighted shortest job first." Business value plus time criticality plus risk reduction value. 24(2) The cost of delaying a project is a risk. 24(3) The unit value of the test case is determined by how the test case ensures that the software will be delivered quickly. 24(4) The unit value function can be used in both a static and dynamic way. 24(5) The combination of multiple test cases impacts the unit value of each test case, because one test case might be correlated with

		another test case. Our test case value function does not incorporate this factor of this complex situation. 24(9) Re-evaluate the value of an unused test case based on finding that the use of a related test case turned out to be valuable. 25(5) The function needs a business value factor that allows for both the importance of revenue generation by the software and the value of internal facing applications. 25(6) For choosing a test case, managers usually endow a value to the test case based on their working experience and intuition. After implementing the test case, the value maybe changed according to the test result. So the value of a test case is dynamic. 25(9) A use of the function is to justify requests for testing resources. 26(2) The value function should be used to evaluate the value of a test case up front in a static sense. 26(3) The value of a test case can be changed in a dynamic sense over time, but that is the exception rather than the rule. 27(7) 90% of test cases are new test cases for testing new functionality. Before running test cases, senior managers always have a list of test cases in their minds based on their initial expectations of the effect of the test cases. 27(8) The value of a test case may or may not depend on whether it is added to a regression suite up front.
Revenue	The impact of revenue generation resulting from the test case failing or unfailing to find bugs in an application.	3(2) Priority is influenced by the amount of revenue that the software will generate. 3(3) The value of a test case is directly related to the amount of revenue that the software under test is likely to generate. The more revenue the software is likely to generate, the more value the test case possesses. This could be an additional factor in the unit value function. 3(6) Try to remove as much subjectivity as possible from the function. Objectivity can be based at least partly on revenue projections of the software. 5(5) Revenue generation is not a separate factor but is part of priority. 8(7) We need a new factor in the unit value function that considers the revenue generation of the software under test. 10(1a) Priority significantly influences the judgment of unit value. In practice, priority of a test case always associates with revenue generation. 12(2) Although revenue generation is a terrific factor to evaluate the unit value, cash flow is also a valuable factor. In some cases, a

		successful test case ensuring that the application runs normally could result in a large cash flow, which is extremely important for a company to be operating persistently. 17(2) Risk partly depends on the potential revenue that the software will produce. 18(5) The value of the test case depends on the amount of revenue that the software is projected to bring in. 20(1) The business value of an application is the key issue and is more important than the revenue it brings in. Some applications are internal and do not bring in revenue. 21(1) The unit value of a test case depends on the business value of the software. Generating revenue is only part of the value of the software along with other business "options." 25(3) The unit value is a comprehensive concept that presents more than just the revenue generation from the application. 27(4) Revenue generation is a factor in projecting the value of test case, but it is not the only factor to be considered.
Code Coverage	The lines of code tested in a given testing case.	3(5) Code coverage (and therefore application features implemented) should be considered when evaluating the value of a test case. A test case that tests more of the code (and by extension more of the application features) has a higher value than test cases that cover less code. This could be an additional factor in the unit value function. 6(5) The value of a test case is greater if it tests a specific part of the software because it can help locate the source of a defect in the code more easily. Thus, Utility of Test Case could be a new factor to be added in the function. 8(9) To justify what kind of code coverage is great depends on whether the requirement of testing is satisfied rather than whether the code coverage is complicated or simple. 9(7) The purpose of test cases is the most important criterion for justifying whether code coverage is good or bad. 9(8) The issue of code coverage as a factor in the value of a test case depends on what you are trying to accomplish. A test case needs to support the type or level of testing for which it is proposed. 10(7) It is hard to say whether a test case with great code coverage is better than one with small code coverage and vice versa. Each has its advantages. However, a test case in a regression test suite is more valuable than one that is not. 11(5) The utility of a test case is more critical than the code coverage of a test case. Test cases that have a multi-function effect are preferred.

		12(5) There is no significant difference between simple code coverage and complicated code coverage. Whether the code coverage works well is the most important point. 13(11) The amount of software that a test case covers may or may not increase its value. 16(2) Code coverage is part of TR. 18(3) A special, single use test case has greater value if it tests a critical part of the code. 18(4) The value of a test case cannot be determined by the amount of code coverage. 19(4) The code coverage would affect the value of a test case. 20(4) The value of a test case increases with the amount of its code coverage because it helps to reduce the number of test cases. 21(3) The code coverage of a test case is a simple concept that neither indicates the complexity of the test case nor the coverage of tested function. 23(1) A test case that test the code's critical path is more valuable than one that does not. 24(6) Code coverage is not a good measure for projecting the value of the test case. In contrast, the functional coverage is more effective. 25(4) The more of the critical path that test case covers, the more value the test case creates. 27(6) Code coverage is not the only factor to determine the value of the test case. A test case is valuable if it tests any amount of code if that code is a critical part of the application.

Frequency	The amount of use of the test case.	4(3) In practice, the frequency of using a test case is a critical standard in evaluating its value. High usage frequency of a test case always presents greater value and priority compared to test cases with low usage frequency. Is the test case used once or does it become a member of a regression test suite? How often is the regression test suite run? This could become an additional factor in the unit value function. 13(10) The more places in the development cycle a test case is used, the more valuable it is. 18(2) A test case that is used to test multiple versions of software or packages is more valuable. 21(2) Repeatability, meaning whether a test case can be used across different regions, devices, or platforms, is a factor in the value of a test case. High repeatability indicates high value of a test case. 23(7) Test cases that test software across multiple mobile platforms are more valuable.
Litigation	Risks arising from litigation regarding an application.	5(6) Test cases become higher in priority if there is a danger of litigation regarding the software.
Globalization	Risks arising from globalization regarding an application.	5(7) Many factors, for example localization/globalization go into priority. Possibly list these factors in a table.
ROI	The value estimation of a test case from ROI angle.	6(1) Consider a Return of Investment (ROI) approach when considering the unit value of a test case. This entails a relative value by ratio in which the Unit Value (UV) of the function comes out an absolute value. Instead of subtracting the unit cost from the risk factors, consider dividing the risk factors by the unit cost. The numerator and denominator do not have to be of the same units. · Case 1: If UV of test case A is 100 (whole risk 200 – unit cost 100) and UV of test case B is almost 100 (whole risk 100 – unit cost 1), plus ROIs of the two cases are equal, how does a test case stand out via the evaluation approaches? The problem is that the UV is basically the same for both but the numbers are very different. · Case 2: test case A and B have the same unit value as well as ROIs, but the vast distinction between A and B is that A need to spend 100 in unit costs and the return period is very long, but B costs much less and the return period is pretty short. How to demonstrate the time issue in the function in the case of UV and ROI being equal?

APPENDIX D

Code Book v2

Table 19. Code Book v2

Code	Definition	Interview Transcript
Internal Risk	Risks arising from events taking place within the organization.	2(2) They currently consider priority, but do not consider executive pressure, which is another reason for splitting out priority as a separate factor. 2(3) Executive pressure may be about quality or speed of development, or both. 6(2a) An additional factor is urgency, such as the deadline for release. 8(3) Executive pressure in Internal Risk (IR) can go two ways. "Do it well," in which case the unit value of a test case should go up; "Do it fast," in which case the unit value of a test case should go down. 8(4) If a test case encounters resource limitations such as budget, time, or personnel, the value of the test case may appear to decrease even though the effectiveness of the test case is significant. In other words, it's not the test case's fault if the project has run out of time or other resources for it. For this research project, the assumption has to be that if a test case is being considered, there is enough time for it. 8(10) Change the rows in the table for Internal Risk (IR) to time, quality, and cost. 10(4) Both high quality and time pressure apply to IR. 13(1) Need to consider business criticality/mission criticality in Internal Risk (IR). "Tier 1" mission critical system. 13(2) Speed to market and agility are Internal Risk (IR) factors. 15(5) Regarding IR, is the test case important "to someone who matters?" 15(9) IR can change over time, especially in agile development as requirements change. 16(3) Executive pressure in IR could be dependent on the area of the company the application is being developed for. 18(7) Risk factors are assigned at different levels of the company. For example, IR is assigned at the executive level. Testers on the front lines, normally have to follow the high level managers' directions regarding risk factors. 19(7) IR includes time pressure, resource pressure, and opportunity cost.

		20(6) The time dimension is a critical factor that should be added into the function. The unit value of a test case is subject to release time pressure in IR and competitive pressure in ER. There is a tradeoff: Time pressure is a zero-sum game. If there is time pressure it may be more important to have defect-free software but it requires more time to test.
External Risk	Risks arising from the events taking place outside of the organization.	1(5) Look into risk in other areas, such as mergers and acquisitions in the financial field. 5(6) Test cases become higher in priority if there is a danger of litigation regarding the software. 5(7) Many factors, for example localization/globalization go into priority. Possibly list these factors in a table. 6(2b) Regulations, and unique feature with extremely competitive strength in the market. It is appropriate to separate priority from the risks. 6(3) External risk factors influence internal risk factors. 7(1) When considering the priority of test cases, risks such as crucial impacts of failure in production and relevant regulations, as well as other risk items listed but not limited in the table, always influence the priority of a test case. Therefore, priority should not necessarily be listed as a new factor in the function. Priority should be listed as another row in Internal Risk (IR). 7(7) In External Risk (ER), split out crucial versus non-crucial impacts. For example, a failure that impacts revenue is one kind of problem while a usability issue is another kind of problem. 10(3) The regulation issue listed in ER is not parallel with the legislation issue. In some cases, products may not violate applicable laws but could be in violation of conventional regulations. 11(2a) Business impact could impact the priority of the software. This could be an additional line item in TR. 11(3) ER is based on the probability of the software failing in production. 15(8) An issue of ER is impact to the brand. 18(6) The value of a test case depends on the potential loss of customers if the software fails in production. 23(8) Test cases that test software that could affect customer satisfaction are more valuable. 24(8) ER should be changed to an overall production risk (PR) to take into account both external facing and internal facing applications.

		25(1) The ER could be affected by the impact from social media that may influence the potential customers' judgement. 25(2) ER can be considered as a production risk because the application may be either external facing or internal facing. 27(2) Production risk as a general issue is more realistic than what we had been considering as failure in production of an external application.
Technical Risk	Risks arising from technical perspectives regarding the application.	2(6) The function also should be considered from vendor group perspective rather than only from testing group perspective. "Vendor issues" should be a line item in TR. 3(7) Add History of the SUT as an additional Technical Risk (TR) line item in the table. 3(8) In the Technical Risk (TR) table, replace "dependency issues" with the degree to which this piece of software affects or interacts with other pieces of software. 4(4) Maintaining the line items of Technical Risk (TR) in a table rather than putting each of them separately in the unit value function is a good idea. 7(2) Since test cases always encounter various kinds of Technical Risk (TR) in the software under test, each line item of TR listed in the table should be weighted separately and then combined to form the total TR. Without this there is a degree of inconsistency. This may not be necessary for IR and ER. 9(5) The time factor, which often influences a decision maker's judgment, should be taken into account in the function as a significant factor. A proposed test case which, based on history, is projected to run in a shorter amount of time is more valuable. 9(6) An automated test case is more valuable than a manual test case. 10(6) All the items listed in TR help people from diverse perspectives evaluate the confidence of the testing staff for completing a test case well. Different testing groups in different situations, however, perceive different TR line items to be more or less important. Thus, the TR line items should be weighted separately to allow for the needed diversity. 11(2b) level of change of the application are two dimensions that could impact the priority of the software. This could be an additional line item in TR. 11(7) Another line item in TR should be the complexity of database interfaces and issues of the application being in the cloud. 12(4) In his situation, complexity and dependency of the software are more critical than the other items of TR. However, the critical level of the items could vary in different situations, so having TR line items with separate weights is a good solution.

		13(7) Difference in the value of a test case based on whether it's new or if we have experience with it. 16(4) The newness of the testing technology, either to the industry or to the company, is a factor in TR. 16(5) Cooperation in different departments or groups is common and is a factor in TR. 17(4) It is better to break down the TR into several subcategories, each of which has its own weight. 19(8) We need more detail for each TR factor, including individual weights. 21(11) A new line for TR is the complexity of the test data. 21(12) The previous testing line in TR includes "brittle code." 21(13) TR includes badly designed test cases. 21(14) A poor test environment doesn't make a test case more or less valuable. 21(15) Does the "test environment issues" line in TR belong there? 22(6) The TR line items should have individual weights. 22(7) The TR line items should be promoted into the main function. 22(8) Another TR line item should be architectural complexity, e.g. asynchronous versus synchronous web service calls. 23(9) The line items in TR should be weighted separately. 25(7) TR is also influenced by the project, which can be called project risk. Project risk is normally caused by limited resources, such as having a fixed date by which the project must be completed. 26(4) Another line item for TR is if the code comes in late to the testers. 27(3) Lacking staff or other resources such as servers because of a delay in acquiring ordered hardware are TR factors. 27(5) Regime change, i.e. changing from full-time employees to contractors is a TR factor. 27(9) The individual line items of TR should be weighted individually.
Amount of Use	The amount of use of the application under testing. The comments discussed the relationship	1(3) The amount of use is a very critical factor. Multiplying external risk by it might be insufficient to elaborate its important role. Multiplying the entire set of risk factors is an option for this point. Different people may have different views on the importance of amount of use.

	between amount of use and other factors.	1(4) The special case, high risk but low amount of use, should be demonstrated. 2(5) Amount of use, AU, is an ideal concept for estimating the importance of a test case. It may be easier to estimate AU for a customer-facing application than for an internal application. 2(7) Uncertain about whether AU should be expanded to multiply more risk factors or not. 2(8) The estimate of AU varies depending on whether the software is a new application or a revision of an existing application. 3(4) The amount of use is an expected and subjective number, not an accurate number in practice. Multiplying the external risk by the amount of use makes more sense than multiplying the entire risk by the amount of use. 4(2) It is better to multiply the entire risk by the amount of use (AU) rather only multiplying the external risk by the amount of use. 5(3) Multiplying only ER by AU is appropriate in the function. 6(4) It makes sense to use AU for calibrating the UV of a test case. But there are several options (listed below) for AU to combine with other factors. The AU could be considered as the probability of software failure. (IR+ER)*TR*AU/UC IR+(ER*TR*AU)-UC (IR+ER+TR)*AU-UC 7(3) It is appropriate to include AU as it currently appears in the function. It applies to External Risk (ER), not to the other risk factors. 8(8) AU is particularly relevant to External Risk (ER.) 8(11) External Risk (ER) is the only factor that should be multiplied by Amount of Use (AU). 10(1b) Customer satisfaction, amount of use, etc. In a word, all of those perspectives are subject to or related to revenue generation, including not overcharging or undercharging customers. 10(5) ER multiplied by AU is preferable, since AU is much more directly correlated with ER than with the remaining factors. If the software fails, you are going to lose revenue or customers. 11(4) Multiplying ER by AU is better than multiplying the other factors by AU. 12(3) AU Multiplying only ER, rather than the other factors, by AU, is correct. 13(4) Leans towards multiplying all three risk factors by Amount of Use (AU).

		14(1) Agree with the AU multiply by ER. If the software has higher customer rate, it implies the software would be used much more frequently and the highest risk is if it affects the most customers. 15(1) AU is related to some aspects of ER rather than all of the elements of ER. For instance, relevant regulations are part of ER but are not affected by AU, while other aspects of ER are affected by AU. 15(6) Multiply only ER by AU, not the other risk factors. 16(1) Multiply all risks by AU. 17(1) The AU should multiply all three risk factors: IR, ER, and TR. 18(8) AU influences the value in revenue potential. This is separate from IR and ER. 18(9) The insight for the function: (IR+ER) [AU-(UC+TR)]. Note that this implies that AU multiplies all risk factors. 19(1) AU should multiply all risk factors: IR, ER, and TR. 19(2) AU may not always be a good indicator of the value of a test case. One-time use of an application could be critical and present a very high risk if the software fails. For example, signing-up a new customer could mean the loss of the customer if the application fails. This also implies that AU and revenue generation may not directly correlate. 20(2) AU should multiply all of IR, ER, and TR. 20(8) AU could mean how many new customers need to be signed-up, not just how many times the customers use the application. 22(1) AU should only multiple ER. 22(2) AU of the critical path is crucial. You have to be careful about multiplying ER by AU because not all features of the software might be used all of the time. 23(5) The IR of a piece of software increases if its use goes across multiple divisions within a company. Therefore, test cases that test this software are more valuable. 23(6) External risk increases if the software is intended to be used in many countries on a global basis. Therefore, test cases that test this software are more valuable. 23(10) AU should be associated with “failure costs” in ER. AU should also multiply TR. 24(7) Different parts of the code may be important for different reasons. The code for particular exception conditions may be very important even if infrequently used. Therefore AU is only one of the influential factors representing the critical level of the software. In

		some cases, software may have very low AU but still have a very high critical level which cannot be ignored. 26(1) AU should multiply both IR and ER. 27(1) AU multiplying just ER or multiplying all risk factors could go either way.
Code Coverage	The lines of code tested in a given test case.	3(5) Code coverage (and therefore application features implemented) should be considered when evaluating the value of a test case. A test case that tests more of the code (and by extension more of the application features) has a higher value than test cases that cover less code. This could be an additional factor in the unit value function. 6(5) The value of a test case is greater if it tests a specific part of the software because it can help locate the source of a defect in the code more easily. Thus, Utility of Test Case could be a new factor to be added in the function. 8(9) To justify what kind of code coverage is great depends on whether the requirement of testing is satisfied rather than whether the code coverage is complicated or simple. 9(7) The purpose of test cases is the most important criterion for justifying whether code coverage is good or bad. 9(8) The issue of code coverage as a factor in the value of a test case depends on what you are trying to accomplish. A test case needs to support the type or level of testing for which it is proposed. 10(7) It is hard to say whether a test case with great code coverage is better than one with small code coverage and vice versa. Each has its advantages. However, a test case in a regression test suite is more valuable than one that is not. 11(5) The utility of a test case is more critical than the code coverage of a test case. Test cases that have a multi-function effect are preferred. 12(5) There is no significant difference between simple code coverage and complicated code coverage. Whether the code coverage works well is the most important point. 13(11) The amount of software that a test case covers may or may not increase its value. 16(2) Code coverage is part of TR. 18(3) A special, single use test case has greater value if it tests a critical part of the code. 18(4) The value of a test case cannot be determined by the amount of code coverage. 19(4) The code coverage would affect the value of a test case.

		20(4) The value of a test case increases with the amount of its code coverage because it helps to reduce the number of test cases. 21(3) The code coverage of a test case is a simple concept that neither indicates the complexity of the test case nor the coverage of tested function. 23(1) A test case that test the code's critical path is more valuable than one that does not. 24(6) Code coverage is not a good measure for projecting the value of the test case. In contrast, the functional coverage is more effective. 25(4) The more of the critical path that test case covers, the more value the test case creates. 27(6) Code coverage is not the only factor to determine the value of the test case. A test case is valuable if it tests any amount of code if that code is a critical part of the application.
Test Frequency	The amount of use of the test case.	4(3) In practice, the frequency of using a test case is a critical standard in evaluating its value. High usage frequency of a test case always presents greater value and priority compared to test cases with low usage frequency. Is the test case used once or does it become a member of a regression test suite? How often is the regression test suite run? This could become an additional factor in the unit value function. 13(10) The more places in the development cycle a test case is used, the more valuable it is. 18(2) A test case that is used to test multiple versions of software or packages is more valuable. 21(2) Repeatability, meaning whether a test case can be used across different regions, devices, or platforms, is a factor in the value of a test case. High repeatability indicates high value of a test case. 23(7) Test cases that test software across multiple mobile platforms are more valuable.
Weight	The weight of factors in the function.	1(6) Write a description of how to use the weights in the function. 14(5) Weights should be different for each factor because each item has different effect and risk in different context. 21(7) Weights can go to zero if a factor is not important.
Unit Cost	Comments regarding the cost of a test case.	2(4) Unit cost is always evaluated in terms of dollars, especially in those departments concerned with preparing budgets. Those departments usually can estimate a relatively accurate estimate of the unit cost based on historical records. 15(4) Breaking down UC into two categories: one time costs (preparation costs, creation costs) and multiple times costs (run costs, failure cost), would make clear sense.
Dynamic Function	Value of test case may vary after execution.	5(2) The Unit Value of a test case is projected before application execution rather than after the process.

		14(9) Recursive use of the value function – do you find greater value after you start running the test case because, for example, it finds a lot of defects. 15(2) The value of a test case may change as you use it in testing. There is a scenario that is pervasive in practice. A test case that was considered to be low value in the initial stage may increase in value due to more defects being detected after the test case execution. 15(3) The value of a test case may increase as you use it in testing as you realize that the code it is testing is more complex than originally thought. 15(7) The value of a test case changes over time and so value should be considered to be in a feedback loop. 19(5) The value function should be considered as a dynamic function rather than a static function because the value of a test case may change after the test case is executed. 19(9) The entire value model should be dynamic because everything can change, "in a heartbeat." 21(5) Especially for a new system where you're not sure about the critical path, the unit value of a test case function could be dynamic. 24(4) The unit value function can be used in both a static and dynamic way. 25(6) For choosing a test case, managers usually endow a value to the test case based on their working experience and intuition. After implementing the test case, the value maybe changed according to the test result. So the value of a test case is dynamic. 26(2) The value function should be used to evaluate the value of a test case up front in a static sense. 26(3) The value of a test case can be changed in a dynamic sense over time, but that is the exception rather than the rule.
Unit Value	General comments regarding the component and structure of the value function.	3(1) The unit value may be positive or negative, it depends on whether the risk value is greater than the cost. 5(4) There is no difference in the value of a test case whether it tests a small piece of software or a large piece of software. 5(8) Keep the unit value function simple while providing a table with details below it. 7(5) A test case that tests a series of applications is more valuable than one that does not. 8(5) The meaning of unit value is to explain why one test case should be implemented versus another. 9(3) Some items of Internal Risk (IR) and External Risk (ER) should be clarified, such as "crucial impacts of failure in production".

		9(9) The definition of unit value should be clarified. In different contexts, it may be comprehended as customer satisfaction-oriented, or revenue generation-oriented, or margin increase-oriented, or other related perspectives. The value of a test case depends on the context in which it is used. 9(10) Comparing the value of different groups of test cases is future research. 9(11) We have to define what we mean by "value." 13(3) If this is a good test case, the testing process will be better. 14(6) In practice, choosing a test case among several choices depends on good guess or experience the test group has. The value function is a good tool for testing people in selecting an appropriate test case in terms of the value. 14(8) Using this value function will help prioritize the work to make the products better. 19(6) A test case is more valuable if it is used in end-to-end testing. 20(3) We are not comparing adding a test case at the unit level to another level. 20(7) Risk values should not be on a linear scale but should be on an exponential, modified Fibonacci scale. Doing this may eliminate or reduce the need for weights. 21(6) Risk values should be on an exponential scale. 22(3) The value of a test case should be a factor of producing revenue or reducing cost. 22(9) Risk values should be on an exponential scale. 23(2) A test case is more valuable if it tests an application in such a way that it makes sure that applications that are communicate with it are not adversely affected. 23(3) A test case is more valuable if it covers multiple countries that an application is intended to be used in. 23(4) The value of a test case is based on the business value of the software under test. 24(1) ScaledAgileFramework.com (SAFe) orders the development of software as the "weighted shortest job first." Business value plus time criticality plus risk reduction value. 24(2) The cost of delaying a project is a risk. 24(3) The unit value of the test case is determined by how the test case ensures that the software will be delivered quickly.

		24(5) The combination of multiple test cases impacts the unit value of each test case, because one test case might be correlated with another test case. Our test case value function does not incorporate this factor of this complex situation. 24(9) Re-evaluate the value of an unused test case based on finding that the use of a related test case turned out to be valuable. 25(5) The function needs a business value factor that allows for both the importance of revenue generation by the software and the value of internal facing applications. 25(9) A use of the function is to justify requests for testing resources. 27(7) 90% of test cases are new test cases for testing new functionality. Before running test cases, senior managers always have a list of test cases in their minds based on their initial expectations of the effect of the test cases.
Test Utility	The effect of the test case.	13(8) Quality of a test case is important and is based on the number of defects found by it. 14(3) A test case that finds no defect is just as valuable as one that finds defects. This has no effect on the value of a test case. 16(8) The value of a test case increases if it detects defects in risky code. 16(9) "After the fact" increases in test case value can occur if the test case finds defects. This could cause you to decide to add it to the regression suite. 22(4) The value of a test case should be based on the function points (i.e. requirements) instead of the amount of code or of specific parts of the code covered.
Regression Suite	Value of the test case may or may not associate with incorporating in regression suite.	5(9) Future research: What is the value of continuing to have a test case in a regression suite? 7(4) A test case that goes into a regression suite is more valuable than one that does not. 13(9) A test case in a regression suite is more valuable than one that is not. 14(2) Adding a test case to the regression suite is not a necessary condition for evaluating the value of the test case. A special test case that is specifically targeted for a reason and used once can be just as important as a test case that goes into a regression suite. 16(7) Normally, they would add a test case into the regression suite unless it's too complex to run. This is not a matter of the test case's value. 17(3) The value of a test case increases somewhat if it is added to a regression suite.

		18(1) Whether the test case is eligible to be added into a regression suite cannot significantly affect the value of the test case because all test cases are added to a regression suite. 19(3) If a test case is added into a regression suite, it indicates that the test case has higher value than test cases that are not added to a regression suite. 20(5) The value of a test case is determined up front and a high-value test case is added into the regression suite. Value is not determined by the decision of whether or not to add it to the regression suite. 21(4) Adding a test case into a regression suite could be the standard to evaluate the value of a test case. 21(9) A test case that is targeted to a part of an application is just as valuable as a test case that goes into a regression suite. 21(10) Another use of the unit value of a test case function is to reevaluate the test cases in an existing regression suite. 22(5) The most valuable test cases are the ones that go into the regression suite. 27(8) The value of a test case may or may not depend on whether it is added to a regression suite up front.
Revenue Generation	The impact of revenue generation resulting from the test case failing or unfailing to find bugs in an application.	3(2) Priority is influenced by the amount of revenue that the software will generate. 3(3) The value of a test case is directly related to the amount of revenue that the software under test is likely to generate. The more revenue the software is likely to generate, the more value the test case possesses. This could be an additional factor in the unit value function. 3(6) Try to remove as much subjectivity as possible from the function. Objectivity can be based at least partly on revenue projections of the software. 5(5) Revenue generation is not a separate factor but is part of priority. 8(7) We need a new factor in the unit value function that considers the revenue generation of the software under test. 10(1a) Priority significantly influences the judgment of unit value. In practice, priority of a test case always associates with revenue generation. 12(2) Although revenue generation is a terrific factor to evaluate the unit value, cash flow is also a valuable factor. In some cases, a successful test case ensuring that the application runs normally could result in a large cash flow, which is extremely important for a company to be operating persistently.

		17(2) Risk partly depends on the potential revenue that the software will produce. 18(5) The value of the test case depends on the amount of revenue that the software is projected to bring in. 20(1) The business value of an application is the key issue and is more important than the revenue it brings in. Some applications are internal and do not bring in revenue. 21(1) The unit value of a test case depends on the business value of the software. Generating revenue is only part of the value of the software along with other business "options." 25(3) The unit value is a comprehensive concept that presents more than just the revenue generation from the application. 27(4) Revenue generation is a factor in projecting the value of test case, but it is not the only factor to be considered.
Simplicity	To simplify the function with main factors.	1(1) To keep the function simple, expand out TR and UC in accompanying tables. 7(6) Do not promote any line items in the table into the main function. It would get too complicated. 8(6) Keeping function simple would help people readily comprehend the meaning of the function. Do not promote table rows into the function. 10(8) Leaving the function with a limited number of factors while providing the table below it with further details is good. 13(5) Leave the main function as it is. Don't expand any of the factors in the main function. 13(6) Leave the function as multiple factors; don't try to combine them. 14(7) Do not expand TR and UC in the function. 16(6) It is not necessary to expand the UC factor in the main function. 21(8) Too much detail in the function could be confusing. 25(8) Keeping the equation simple without promoting the TR line items to the equation is a good way to present the unit value of a test case.

Priority	A factor has relative higher or lower level of importance against another factor.	1(2) Risk is not equal to priority. The priority should be considered as a separate factor in the function. 2(1) A priority factor should be added in the function. Priority might be rated as high, medium, and low. 4(1) The priority of a test case is influenced by how critical the application feature is that the software is implementing, such as handling customer complaints, dealing with cutting edge technologies being used by competitors, etc. 5(1) The priority of a test case should be a separate factor. But the relationship between revenue generation and priority is very limited, since it is hard for a testing group to figure out the amount of revenue generated by a feature or application. 8(1) Priority always accompanies risk, especially in internal risk and external risk. It is hard to list priority separately. The standards of priority vary from one case to another. Sometimes satisfying customers, which is an element of external risk, is prioritized. In some other situations, release time, which is an element of internal risk, is considered the most critical element. 8(2) Every risk is a matter or measure of priority. They are two sides of the same coin. 9(1) Risk significantly differs from priority. Priority is an aspect of risk. Priority varies in different contexts. Priority might focus on the number of customers affected or what would happen to the brand if a problem hit the media. 9(2) Priority is a piece of both Internal Risk (IR) and External Risk (ER). 9(4) Death is a top priority. Safety is a top priority. 10(2) The priority of a piece of software falls if there are work-arounds that render the software unnecessary. 11(1) The priority of a test case is not the same as the unit value of a test case, but they are closely related to each other. 12(1) Priority can be based on different reasons. How important is the software to the customer, to the business, or to marketing efforts? 14(4) Priority is a part of risk; high priority leads to high risk. For example, the media attention of the software would raise the ER of the test case.

ROI	The value estimation of a test case from ROI angle.	6(1) Consider a Return of Investment (ROI) approach when considering the unit value of a test case. This entails a relative value by ratio in which the Unit Value (UV) of the function comes out an absolute value. Instead of subtracting the unit cost from the risk factors, consider dividing the risk factors by the unit cost. The numerator and denominator do not have to be of the same units. · Case 1: If UV of test case A is 100 (whole risk 200 – unit cost 100) and UV of test case B is almost 100 (whole risk 100 – unit cost 1), plus ROIs of the two cases are equal, how does a test case stand out via the evaluation approaches? The problem is that the UV is basically the same for both but the numbers are very different. · Case 2: test case A and B have the same unit value as well as ROIs, but the vast distinction between A and B is that A need to spend 100 in unit costs and the return period is very long, but B costs much less and the return period is pretty short. How to demonstrate the time issue in the function in the case of UV and ROI being equal?