INTELLIGENT EVALUATION SYSTEM FOR SOFTWARE QUALITY MEASUREMENT

BY

NWANDU, IKENNA CAESAR

B.Sc [UNIBEN], M.Sc [FUTO]

(20164023638)

A DISSERTATION SUBMITTED TO

THE SCHOOL OF POSTGRADUATE STUDIES

FEDERAL UNIVERSITY OF TECHNOLOGY, OWERRI

DECEMBER, 2022

INTELLIGENT EVALUATION SYSTEM FOR SOFTWARE QUALITY MEASUREMENT

BY

NWANDU, IKENNA CAESAR

B.Sc [UNIBEN], M.Sc [FUTO]

(20164023638)

A DISSERTATION SUBMITTED TO

THE SCHOOL OF POSTGRADUATE STUDIES

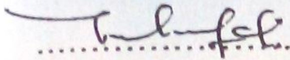
FEDERAL UNIVERSITY OF TECHNOLOGY, OWERRI

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF PhD DEGREE IN COMPUTER SCIENCE

DECEMBER, 2022

CERTIFICATION

This is to certify that this dissertation titled "INTELLIGENT SOFTWARE EVALUATION SYSTEM FOR SOFTWARE QUALITY MEASUREMENT" was carried out by NWANDU, IKENNA CAESAR (**20164023638**) in partial fulfilment of the requirements for the award of the degree of Doctor of Philosophy (PhD) in the department of Computer Science, Federal University of Technology, Owerri, Imo State, Nigeria.

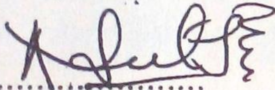


Dr. (Mrs) J. N. Odii
Supervisor

14/11/2022
Date

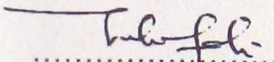
Dr. (Mrs) E. C. Nwokorie
Co-Supervisor

14/11/2022
Date



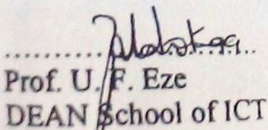
Dr. S. A. Okolie
Co-Supervisor

14/11/2022
Date



Dr. (Mrs) J. N. Odii
Ag. H.O.D. Computer Science

14/11/2022
Date



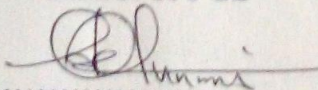
Prof. U. F. Eze
DEAN School of ICT

14/11/22
Date

....

Prof. B. O. Esonu
DEAN School of PGS

....
Date



Prof. A. O. Adetumbabi
External Examiner

14/12/22
Date

DEDICATION

This thesis is dedicated to:

My father, Sir Athanasius Nwandu of blessed memory,

&

My mother, Lady Josephine Nwandu.

ACKNOWLEDGEMENTS

I express my inestimable gratitude to God almighty for the unmerited grace he bestowed on me especially during the course of this program.

My immeasurable thanks go to Emeritus Prof. E. O. Nwachukwu whose supervisory expertise brought this work to shape. I am most grateful to the Head of Computer Science Department who is also my principal supervisor, Dr. (Mrs) J. N. Odii for her ever-readiness to attend to me throughout the period of this work and for expediting the completion of this program. I highly appreciate the efforts of my co-supervisors Dr. (Mrs) E. C. Nwokorie and Dr. S. A. Okolie for giving their succinct but appropriate corrections when needed. I will never forget the reviewer of this work Dr. C. O. Ikerionwu for his meticulous efforts to read in between the lines and enforced corrections that brought this work to an acceptable standard. My appreciation goes to the Deans of Schools of ICT and Postgraduate Studies, Prof. (Mrs.) U. F. Eze and Prof. B. O. Esonu as well as the immediate past Dean PGS Prof C. C. Eze for their lenient responses whenever they are approached. I remain grateful to my lecturers especially Professors E. N. Erumaka, P. O. Asagba and S. C. Inyama; Doctors C. N. Njoku, C. G. Onukwugha, C. L. Okpalla, C. Anyiam, I. I. Ayogu, O. A. Njoku, M. E. Benson-Emenike, U. Onyemauche, T. U. Onwuama and all other lecturers in the department of Computer Science and the school of ICT at large whose scrutiny and contributions metamorphosed this thesis into a masterpiece.

I specially appreciate my dear mother, Lady Josephine Nwandu, my brother, Rev. Fr. Dr. Paul Nwandu, my sister Victoria, my wife Chinwe and my other concerned siblings for their encouragement and numerous supports. I extend a post-humus appreciation to my late brother, Chief Elias Nwandu whose lifestyle inspired me to pursue higher education, may his soul rest in peace. I will not fail to thank my colleagues: Francisca Nwokoma and Donatus Njoku for their spirit of teamwork. Finally, I appreciate the efforts of my friends, so numerous to be mentioned, for the diverse aids rendered to me during the course of this work and the entire program.

ABSTRACT

The concern about the large-scale and complexity of contemporary software cannot be over-emphasized. This is inclined to the assurance of standardized software quality which is essential for preventing disastrous effects of releasing fault-prone systems. This thesis designed an intelligent model that uses various metrics corresponding to six quality attributes (namely Reliability, Usability, Efficiency, Functionality, Maintainability and Portability) to measure the quality of software. This agreed to the assertion that software quality evaluation process is an instrument that observes the characteristics of a software product. In software engineering, the primary quality evaluation and assurance technique that establishes confidence over successful execution of software is termed software testing. Software testing usually identifies and applies metrics to software products in order to promote and assess their quality. This thesis designed an intelligent evaluation model in conformance with software testing principles. The objective of the model is to apply reinforcement learning in its software evaluation process to measure six software attributes in terms of speed of execution and to ensure optimal decision-making in the evaluation process, such that the model returns a reliable outcome. The model utilized a formulated model equation, whose input are the measured attributes, to achieve the evaluation. The model is developed using extreme programming principles, an agile framework whose operation is based on simplicity. It also adopted object-oriented analysis and design methodology which allowed the utilization of various artifacts including use cases, data flow, sequence, flowchart, entity-relationship and class diagrams to describe the architecture and functionality of the system. The model was implemented using Python programming language with the database design on MySQL platform. The model is further validated by comparing its performance measures on test data gotten from the functional information of Oil-palm Management Program and Estate CanePro. These tests data produced quality values of 0.9 and 1.0 respectively via the model equation. These results gave the indication that the resource software perform efficiently owing to the fact that the model's value benchmark is best as it approaches unity. The result of comparing the outcomes showed that reinforcement learning makes software evaluation dynamic and precise. The results indicated that the model independently determines the strategies to follow during evaluations and the same set of data consistently gives the same outcome. The result also showed that the reliability of a software is directly proportional to its usability and maintainability. However, the result also showed that having a high portability value does not guarantee the reliability and/or maintainability of a given software.

Keywords: Software quality, evaluation, measurement, metrics, attributes, testing, reinforcement learning

Table of Contents

Cover Page i
Title Page ii
Certification iii
Dedication iv
Acknowledgements v
Abstract vi
Table of Contents vii
List of Tables xi
List of Figures xii
CHAPTER ONE: INTRODUCTION
1.1 Background Information 1
1.2 Problem Statement 7
1.3 Objectives 9
1.4 Justification of Study 9
1.5 Scope of Study 10
CHAPTER TWO: LITERATURE REVIEW
2.1 Conceptual Literature 11
2.1.1 Software Quality 11
2.1.2 Software Measurement and Evaluation 14
2.1.2.1 Software Measurement 15
2.1.2.2 Software Evaluation 16
2.1.2.2.1 Software Testing in Software Evaluation 28
2.1.3 Software Intelligence 30
2.2 Theoretical Literature 31
2.2.1 Software Evaluation Criteria 31
2.2.2 Classification of Evaluation Models 35
2.2.3 Disadvantages of Conventional Evaluation Models 37

2.3 Empirical Literature 38
2.3.1 Software Evaluation Metrics 38
2.3.2 Existing Software Evaluation Models 52
2.4 Related Work 77
2.5 Direction of the Research 96
CHAPTER THREE: METHODOLOGY
3.1 Methodology of the study 98
3.2 Analysis 100
3.2.1 Analysis of Study Models 100
3.2.1.1 Algorithm of Bindal (2013) Model 103
3.2.1.2 Limitations of Bindal (2013) Model 103
3.2.1.3 Algorithm of Chen *et al*. (2011) Model 105
3.2.1.4 Limitations of Chen *et al*. (2011) Model 106
3.2.1.5 Algorithm of Rana *et al*. (2015) Model 108
3.2.1.6 Limitations of Rana *et al*. (2015) Model 108
3.2.1.7 Algorithm of Nwandu and Asagba (2017) Model 111
3.2.1.8 Limitations of Nwandu and Asagba (2017) Model 112
3.3 Design 112
3.3.1 System Modelling 113
3.3.2 System Model Architecture 114
3.3.3 Advantages of the System 115
3.3.4 Expectations of the System 116
3.3.5 Assumptions of the System Model 116
3.3.6 Performance Measures of the System Model 116
3.3.7 Use Case Diagram of the System Model 125
3.3.8 Data Flow Diagram of the System Model 126
3.3.9 Sequence Diagram of the System Model 127

3.3.10 Flowchart of the System Model 129
3.3.11 Implementation Algorithm of the System Model 131
3.3.12 Input/Output Specifications 133
3.3.13 Entity Relationship Diagram of the System Model 134
3.3.14 Database Specifications 135
3.3.15 Class Diagram of the System Model 137
3.3.16 High Level design of the System Model 139
3.4 Schematic Diagram of the System Model 140
3.5 Programming Language Used 142
3.6 Test Data for Model Validation 142

CHAPTER FOUR: RESULTS AND DISCUSSION

4.1 Hardware Specification 143
4.2 Software Specification 143
4.3 Input Interface 143
4.4 Output Interface 145
4.5 Results 147
4.5.1 Result Analysis 148
4.6 Discussion of Result 150
4.6.1 Realization of objective (i) 151
4.6.2 Realization of objective (ii) 152
4.6.3 Realization of objective (iii) 153
4.6.4 Realization of objective (iv) 153
4.7 Comparison of Proposed Model with Existing Systems 155

CHAPTER FIVE: CONCLUSION AND RECOMMENDATIONS

5.1 Summary 158
5.2 Contribution to Knowledge 159
5.3 Conclusion 159

5.4 Recommendations 160

REFERENCES 162

APPENDIX A: VALIDATION DATA 172

APPENDIX B: PROGRAM CODE 174

LIST OF TABLES

Table 2.1 Sources of Evaluation Criteria for Software Products 34

Table 2.2 Summary of Related Works 89

Table 3.1 Input Specifications 133

Table 3.2 Output Specifications 134

Table 3.3 User Table 136

Table 3.4 Server Table 137

Table 4.1 Output of Built-in Metrics using SW1 Data 147

Table 4.2 Output of Built-in Metrics using SW2 Data 147

Table 4.3 Quality Quantification 148

LIST OF FIGURES

Figure 2.1 Software Quality Interest Groups	13
Figure 2.2 Technical Approach to SQE of Any Software Product	21
Figure 2.3 Processes in the SQE Technical Approach	22
Figure 2.4 Major Software Testing Elements	28
Figure 3.1 Framework of Bindal's System	101
Figure 3.2 Flowchart of Bindal's System	102
Figure 3.3 Framework of Chen *et al.*'s System	104
Figure 3.4 Flowchart of Chen *et al.*'s System	105
Figure 3.5 Framework of Rana *et al.*'s System	107
Figure 3.6 Flowchart of Rana *et al.*'s System	107
Figure 3.7 Framework of Nwandu and Asagba's System	109
Figure 3.8 Flowchart of Nwandu and Asagba's System	110
Figure 3.9 Architecture of the Model	115
Figure 3.10 The Reinforcement Learning Model	122
Figure 3.11 Use Case Diagram of the Model	125
Figure 3.12 Data Flow Diagram of the Model	126
Figure 3.13 Sequence Diagram of the Model	128
Figure 3.14 Flowchart of the Model	130
Figure 3.15 Model's E-R Diagram	135
Figure 3.16 Model's Class Diagram	138
Figure 3.17 Detailed Architecture of the Model	139
Figure 3.18 Model's Schematic Diagram	141
Figure 4.1 Input Interface on Jupyter Notebook Environment	144

Figure 4.2 MySQL Workbench Output Interface 145

Figure 4.3 MySQL Command Line Output Interface 146

Figure 4.4 Graphical Representation of SW1 Metric Values 149

Figure 4.5 Graphical Representation of SW2 Metric Values 150

Figure 4.6 Comparative Representation of the Quality Attributes 154

CHAPTER ONE

INTRODUCTION

1.1 Background Information

The quality of software systems is focused on engineering techniques for developing and maintaining systems. This software quality in discourse is obviously felt through the overall system performance with respect to the engineering techniques that were utilized. In other words, software quality continues to be a fundamental element in the discipline of software engineering (Kassie and Singh, 2020). The IEEE standard glossary gave a clear description of software engineering as a discipline which applies a systematic, disciplined, quantifiable techniques to the design and developmental processes of a software as well as its maintenance culture (Sommerville, 2016). Software products which are the final results of software engineering are fast growing in number and complexity to the extent that there are no human activities that software has not found application (Miguel *et al.,* 2014). Hence, continuous improvement is a basic factor for endurance in today's raging business environment. This counts for the fast-paced world of information technology as well as software engineering itself. Crucial for the improvement of the process is to gain information which enables both the developers and the users to evaluate the state of the process and its products. From the status information, developers

and other stakeholders can plan actions for improvement and also evaluate the success of those actions especially on the state of the resultant product's quality. The quality of software products, as described by the IEEE Standard Glossary of Software Engineering Terminology, is:

i. The degree to which a system, component or process meets specified requirements and
ii. The degree to which a system, component or process meets the needs or expectations of a user.

While trying to formulate a satisfactory definition for a software product, Lai and Brinkkemper (2007) described software product as an aggregation of "software component configuration" or a "software-based service" which contains supporting components with the aim of being released and exchanged to a target market. By aggregated components, Lai and Brinkkemper (2007) referred to the several programs modules that make up the software. Generally, software products take different forms. They can be categorized as:

i. small,
ii. COTS (Commercial Off-The-Shelf) components,
iii. packed software,
iv. large commercial software,
v. open source software and services.

Software evaluation is typically an assessment which aims at determining if a given software or a combination of software programs gives highly satisfactory solutions to the user requirements or needs. The general idea of software evaluation is to make a good analysis of the available resources and tools associated with the software. The analysis is meant to determine whether the resources and tools are currently being utilized by the software or they are to be considered for possible addition to programs already in use by the user. Software evaluation therefore serves as an exposition to the usefulness of the products to the potential user. The outcome of software evaluation invariably informs the stakeholders if a combination of other software products would give better solutions. In other words, software evaluation enables a thorough investigation on the software quality status and creates room for potential improvements in order to ascertain specific objectives. This calls for the evaluation of both the design process and the software system itself to ensure the emergence of a reliable software system. Such evaluation gives an indication of how correct a computer program would be in terms of language features and programming paradigms.

A program and/or software is said to be correct if the program or software does not derail from the original intentions of its designers, i.e., being able to perform its functionality according to the specifications provided by the designers and users ab initio. Often times, programmers describe software correctness in

formal terms. Formally correct software is often stated in mathematical terms in the course of proving its correctness beyond reasonable doubts to convince the designers and users about its relative state of error absence. A program is said to be formally correct if there is a strategy of specifying precisely (mathematically) the intended functionality of the software for all possible values of its input. Software correctness therefore points very closely to its quality. This is because software quality depends on its correctness and vice versa. Hence, in the course of this study, the terms “quality” and “correctness” will be used side by side.

Incorporating some sort of machine learning to software evaluation may give greater level of quality assurance. This is because Machine learning creates room for the design and implementation of algorithms that are intelligent in nature whose executions presents a computer as being capable of learning (Ayodele, 2010). Learning allows for the construction of models that accept inputs which propels the production of the desired result and learning from the data. This ideology indicates that learning is concerned with finding statistical consistencies or other patterns in the supplied data.

Machine Learning pays great attention to determining which techniques to be employed to get the computer program itself based on some experiences and pre-existing conditions (Ayon, 2016). Machine Learning applies some

mechanisms which allow for effective capturing, storage, indexing, retrieval and merging of data. It also incorporates what techniques or algorithms would be able to fuse several subtasks into one single operational unit and how their functionality can be tractable (Mohssen *et al.,* 2017). Machine Learning therefore is a tool that would supposedly give an optimal evaluation of software with a higher degree of reliability.

During software evaluation, several factors are meant to be put into consideration. First, software and available hardware resources on the computer or on the user's network is necessarily checked for compatibility. The compatibility test is centred on the operating system required by the software to effectively perform its functions and also manage the memory space that is provided by the system's hardware. This creates room for a possible hardware and memory upgrade to accommodate the software under consideration. Second, the friendly interaction between the proposed software package and other installed applications is yet another considerable factor. For example, if software meant for processing customer orders cannot be integrated to accounting software, there will be need for more manual preparation of invoices. Therefore, integrating the right sets of software programs is a good step towards harnessing essential functions, enabling users to devote quality time to other activities that add more value to their work output.

The essence of software evaluation is ensuring the use of relevant software products at both individual and organisational levels in order to increases operational efficiency rather than introducing additional workloads. Despite the fact that individuals and organisations can conduct evaluation on their own, experts/consultants could be hired for software assessment. The latter option often uncovers issues that need prompt attention, ultimately adding more values to the software quality.

With the rapid growth of the software market, there are often cases of users not getting the desired software quality. As a result, the Joint Technical Committee of the International Organization for Standardization and International Electro-Technical Commission published a set of software product quality standards namely ISO/IEC 25000 (2014), ISO/IEC 25010 (2011), ISO/IEC 25023 (2016), and ISO/IEC 25022 (2016) meant to provide solutions to the contending issues about software quality. These standards specify the characteristics and sub-characteristics of software product quality and their metrics. Despite the high level of standardization, some software experts argue that ISO/IEC standards lacked supporting evidences. The argument was based on some already existing standards that are seen as assertions that have "codified approaches" (e.g. Pfleeger and Atlee, 2010) which must be proved via a rigorous and scientific process. The argument further criticized over reliance on experts' opinions on setting up standards instead of embarking on careful and rigorous empirical

software engineering researches. On the sequel, measurement and evaluation of software products' quality become paramount.

An evaluation program has to reflect specific organizational problems or goals in order to supply data which can be used in finding solutions to the organizational problems that relate to the organizational goals. In other words, the evaluation program can only deliver relevant information if there is an established link between the business goals and the performance measurements. The described situation necessitated the need to study more about the best practices that can be applied to measuring and evaluating software systems to ensure adherence to quality standards. This would reduce the risks of system dysfunction and failures during their active states.

1.2 Problem Statement

A number of software metrics are already in existence. These metrics provide relevant knowledge of the resources, processes and products that are associated with software development and evaluation. These software metrics provide factual and quantitative information about the software being evaluated via a test process. Available literature reveals some significant research gaps in reviewed scholarly studies such as Chen *et al.* (2011) which limited its scope to only four software products, Rana *et al.* (2015) which focused only on quality categorisation, Nwandu and Asagba (2017) whose focus was on estimating the reliability value of software, Li *et al.* (2019) which focused on predictive

detection of fault rather than its measurement based on system performance, Moreno *et al.* (2020) which focused on the quality of requirements, Cowlessur et al. (2020) which conducted a discovery on insufficient study of software quality prediction, and Omri and Sinz (2021) which laid emphasis on maintenance aspect of software quality. These gaps pose the following problems:

i. Existing evaluation models are user-centred in nature and either resource and cost intensive or can only perform a partial measurement of the software process.
ii. Despite the existence of several metrics, previous evaluation models have not made an exhaustive use of them to aid in more comprehensive measurement of software characteristics which is crucial in ascertaining the quality of software that is being rolled out for use.
iii. Hence, there is need to delve into research for a knowledge-based exhaustive measurement system fused with relevant test metrics that can provide a positive difference in controlling the current deficiencies in this context.

It is essential that some innovative efforts should be directed towards providing a better evaluation system that would bridge the gap presented in the reviewed existing studies. This is essential owing to the increasing rate of software dependency in the conventional life. On that note, this study is poised with the

intention of applying a fusion of machine learning and automation to the investigative process of software measurement and evaluation, relatively different from available models.

1.3 Objectives

The primary objective of this research work is to develop an intelligent evaluation system that measures the quality of software systems.

The specific objectives of the work are to:

i. Design a software quality evaluation model.

ii. Integrate reinforcement learning into the evaluation model such that the system learns from past experience.

iii. Implement test metrics into the model for the measurement of software quality attributes.

iv. Simulate the model and compare its performance measures on various test-data.

1.4 Justification of Study

Intelligent software evaluation would enhance the measurement process of software quality attributes and expression of the qualitative factor by a number. Developers must invest much of their time in quality assurance issues namely reliability, usability, efficiency, functionality, maintainability and portability of software. The proposed model will be appropriate for those experts who desire tools for continuous commitment to making improvements in software

processes. This will ensure the development, deployment and use of software products with satisfactory quality. Thus, serving as an aid to businesses in performing their daily activities more efficiently as well as pinpointing defects in their employed software and make appropriate corrections for the improvement of the overall system capacity and accuracy.

1.5 Scope of Study

This work is limited to evaluating software quality attributes namely reliability, usability, efficiency, functionality, maintainability and portability, using a machine learning-based testing process, realized (simulated) in object-oriented platform.

CHAPTER TWO

LITERATURE REVIEW

2.1 Conceptual Literature

2.1.1 Software Quality

Software quality refers to the tendency of a software system to contain less defects. It has much inclination on the entire software characteristics that gives the idea that the software is capable of satisfying the specified user needs. Notably, the quality of software is deduced from the result of the potential challenges that would be envisaged for the resultant product (Lysne, 2018). Uddin (2022) views software quality as an evident proof of confidence that software can adequately work according to its purpose. It can therefore be said that software quality is a summary of the software's ability to fulfil its intended functionality and the degree of competence at which the functionality is performed. Software quality ought to be managed with proper planning and analysis in order to achieve the needed/target product. It can therefore be viewed as a fundamental factor for the success of software development. Gillies (2011) is credited of a four-tier software quality management (which comprises development procedures, quality control, quality improvement and quality assurance) considered as key in software development.

Software quality is dependent on both the process and the product itself. A software developer takes this into consideration with the intent of promoting the resultant software's quality by ensuring a reliable software development process (Lysne, 2018). However, the quality of software concerns all stakeholders both from the technical and the managerial aspects to ensure that the appropriate methodologies and specifications are duly implemented.

Software quality was introduced in software engineering research for the purpose of ensuring the release of safe and efficient software products (AltexSoft, 2019). The quality of software can be viewed from two basic levels of behavioural expectations and demand namely functional and non-functional (AltexSoft, 2019). The functional aspect of software quality verifies the software product's compliance to the functional (or explicit) requirements and specifications responsible for the practical behaviour of the software. It refers to the software features, performance, user-friendliness, and tendency to be defect-free. On the other hand, the non-functional expectations of software focus on the software's compliance with the structural requirements responsible for the design of software architecture as well as its characteristics. It refers to software efficiency, security, understandability, and code maintainability. The aggregate of the software features and characteristics give a pointer to its quality.

However, there is no best approach to describing software quality due to its complexity (Chappell, 2018). The central idea of software quality is that its evaluation implies the measurement of the values of its attributes which are targeted to be useful to three groups of stakeholders namely users, development team and project sponsors. This is depicted in Figure 2.1. The users subject the software produced by the development team to solution-provision operations whereas the project sponsors' roles are remotely felt through the funding of the software development. The software development is represented as a translational process which transforms an idea into a working software, with the three interest groups showing concern on the quality of the outcome of the development process namely the resultant software.

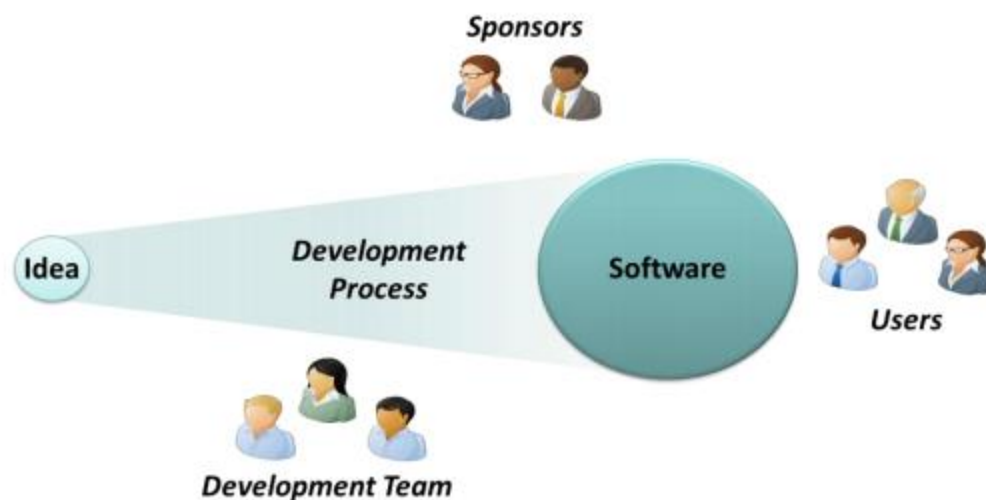


Figure 2.1: Software Quality Interest Groups (Chappell, 2018).

The reality of software quality is of utmost importance especially in this era that harbours systems whose software-intensiveness and heterogeneity on multi-

tenant platforms necessitate the support of multiple users with different requirement demands and behaviour (Tekinerdogan *et al*., 2016). The procedure to be followed to measure the quality of software and the criteria for its evaluation form part of the software quality assurance (Ganney *et al*., 2020).

2.1.2 Software Measurement and Evaluation

The necessity of maintaining improvement culture as regards quality remains a paramount issue in the development of software. The required quality can be achieved by focusing on software's performance, reliability, availability and maintainability. These attributes are closely related to software complexity (Misra *et al.*, 2013). This calls for a continuous effort towards the production of new complexity measures which naturally help to ensure the realization of quality assurance in software processes, software projects and software products (Misra *et al.*, 2013). Owing to this fact, the task of software measurement and evaluation becomes more worrisome. The worrisome concern is believed to have driven software experts into developing techniques to ease the building of good quality software. This development has influenced the creation of models meant for the assessment of software quality (Miguel *et al.*, 2014).

The complexity measure of software product must possess certain characteristics namely validity, sensibility and usefulness. The verification of these properties is often done using evaluation and validation criteria (Misra *et*

al., 2013). In some cases, important decisions such as quality improvement, large-scale procurement, and contracts supervision are analysed using software quality evaluation (Jung *et al.*, 2004). Sequel to this, the construction of software started in the 2000's with large dependency on generated or manufactured components which introduced new challenges to software quality (Miguel *et al.*, 2014). The said components were responsible for the emergence of new concepts including configurability, reusability, availability, better quality and lower cost. A typical software development plan is dynamic and involves some level of uncertainty; every software developer monitors the quality of his software products in relation to the cost of development which gives an insight on how to apply evaluation measures that will controls the quality to fall within the desired level at the required costs. This strengthens the importance of measurement and evaluation in software engineering.

2.1.2.1 Software Measurement

By description, measurement represents attributes of real-world entities in numeric or symbolic forms in a manner that the entities are clearly described with stated standards (Fenton and Bieman, 2014). The entities technically known as attributes are subjected to investigation to expose the manifestation of their sizes, quantity or amount via measurement (Geeksforgeeks, 2020)). In software engineering, measurement is employed as a technique that ensures objective evaluation, different from the popular subjective evaluation

methodologies including interviews, surveys, and inspections. Software measurement is used to enhance the performance of software products or even the process through which they are developed (Geeksforgeeks, 2020). Measurement is meant to provide an understanding of a software process or its resultant products (Tutorialspoint, 2020) usually using appropriate metrics. Measurement of software to encourage quality evaluation however, has limitation because most of the known software metrics are used in the estimation of software costs.

2.1.2.2 Software Evaluation

However, the outcome of software evaluation, i.e., measured quality, is highly dependent on the criteria of evaluation provided by the product users. As a result, software evaluation criteria is expected to be developed by stakeholders in order to reduce the level of uncertainties that may be introduced by some criteria adopted from external sources (Harmon and Metz, 2014).

As a generic phenomenon, evaluation is viewed to possess the following characteristics:

i. Evaluation is a task, whose outcome may be reported as one or more results.
ii. Evaluation is an aid for planning, which means that the result serves as an evaluating parameter to other actions.

iii. Evaluation is a goal-oriented task whose aim is to analyse the results of actions with the intention of either improving the action quality or creating alternative measures that are more suitable for the execution of the action.

iv. Evaluation is subjective to current trends in science as well as modern technological standards.

In software engineering, evaluation is an activity that is concerned with the analysis of software attributes to expose their size, quantity, amount or dimension (Geeksforgeeks, 2020). Software evaluation is an assistive technique in software development whose usage over the years has eventually brought a positive paradigm shift in Human-Computer Interaction (HCI). For instance, the iterative activities of Hix and Hartson's *Star Life Cycle Model* which includes Task analysis, Requirement specification, Conceptual and formal design, Prototyping, and Implementation can be jointly executed in one evaluation process. This is obtainable in most of the common but current software processes such as Scrum and some other agile frameworks (Chappell, 2018). This represents one way in which evaluation is time conserving and also helpful in progressive decision making. Software evaluation can be carried out to analyse different attributes, for example, functionality, reliability, usability, efficiency, maintainability, portability (ISO/IEC, 2011) in order to reveal the characteristics exhibited by the software at a particular stage of development.

This is in line with the assertion that software evaluation is no longer left for the last phase of software development, instead it is an essential task that has to be performed more often in the development process for the purpose of gathering information needed to make progress in the design. This is also evident in the fact that software measurement and evaluation is essential for analysis, predictions and process improvement (Farooq *et al*., 2011).

As stated earlier, every software evaluation that is carried out is usually meant to achieve some objectives. Such objectives can vary but they usually revolve round three question-like assertions (Gediga *et al.*, 2015):

i. Which software gives better solution?

   This portrays evaluation as comparative mechanism which aims at choosing the best software system among several options. This can be seen when selecting the best software tool that is most suitable for a particular application, making a comparison among various versions of a software product to get the best, or making decision among several software prototypes to select the best.

ii. How good is the software?

   The quality level of the software product is emphasised here. This objective therefore evaluates the software product to unleash its usability-goals.

iii. What makes the software bad?

This objective calls for the evaluation of the software product to discover errors or bugs with the intention of identifying corrective measures. This is usually seen during software modification and/or software re-engineering.

Gediga *et al.* (2015) classified the first two objectives "summative evaluation". This is because the two objectives focused on the generic strategies of software development and failed to provide improvement policies that would be applied to make the system better. Summative evaluation comes into play at the near end of software development. However, the third objective was called "formative evaluation", whose emphasis is on improving the software and the design process. This is why developer that are concerned with iterative software development usually adopt this approach (Gediga *et al.*, 2015).

Software evaluation was seen by Farooq and Dumke (2008) to return results in either of two forms namely:

i. qualitative through assessment or
ii. quantitative through measurement.

By qualitative they meant that evaluation follows a subjective reasoning to draw its conclusions. The subjective manner of determining the software's closeness towards meeting requirements is called assessment. Assessment includes carrying out surveys, review exercises, auditing, and system analysis. On the

other hand, quantitative evaluation involves the use of metrics in determining software quality in numerical terms. In this thesis, the quantitative form of evaluation is adopted.

Technically, software quality evaluation (SQE) process can be perceived to be an observation tool which identifies the characterizing features of a software product. SQE as a tool, comprises a skilled evaluator. The evaluator is trained with some evaluation criteria stipulated for this purpose. The duty of the evaluator is to check for any contradiction between the software product and the laid down evaluation criteria. The evaluator raises alarm at the detection of any contradiction, a phenomenon known as anomaly observation. When several anomaly observations have been gathered, the evaluator utilizes an embedded use-risk model to determine the use-risk implications (Harmon and Metz, 2014).

Different evaluator skills and evaluation criteria are required for different software products such as documentation, source code and executable packages. As a result, a database is maintained as a repository of the outcomes of the evaluation. The stored results are used in defining benchmark for controlling the estimation of observation uncertainties. Developers usually tailor software quality evaluation tasks towards satisfying the quality needs of potential users by applying alteration on the conditions of evaluation, the scope of evaluation and the number of experts that will carry out the exercise. They also adopt some

sort of strong project team that will manage the evaluation task in order to get the best results including quality of products and cost involvements (Harmon and Metz, 2014). The implementation of the criteria-based process in the technical approach to software quality evaluation is diagrammatically illustrated in Figure 2.2.

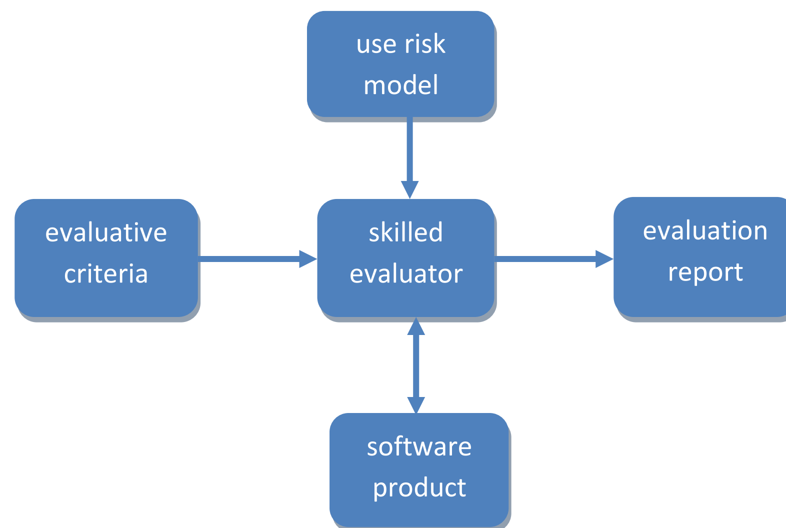


Figure 2.2: Technical Approach to SQE of a Software Product (Harmon and Metz, 2014).

Harmon and Metz (2014) went further to enlist the processes in software quality evaluation technical approach to include:

i. Project management,
ii. Requirements modelling,
iii. Documentation evaluation,
iv. Source code evaluation,
v. Executable evaluation,

vi. Software process evaluation and

vii. Reporting of outputs.

The relationships between these processes are further shown in Figure 2.3.

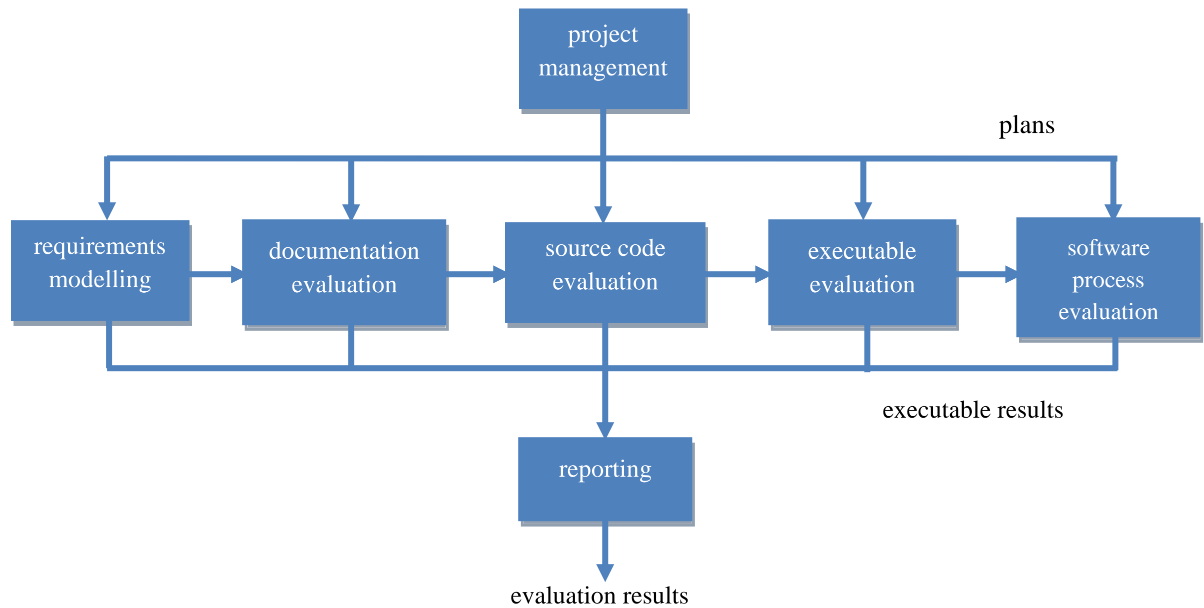


Figure 2.3: Processes in the SQE Technical Approach

(Harmon and Metz, 2014).

In Software Quality Evaluation, requirements modelling is essential (Harmon and Metz, 2014). That is to say that system requirements need to be properly structured to ease evaluating efforts toward ascertaining the system's ability to meet user needs. This is a measure that directly handles the formal analysis of system requirements in a definite manner to encourage risk-based evaluation, evaluation scope assessment and reduction of evaluation uncertainty. Harmon and Metz (2014) are of the view that system requirements can be presented in various ways and that maintaining consistency in representation minimizes the rate at which evaluators bring subjectivity during requirement interpretation.

Evaluating the system documents involves identification of the document quality. Documentation is one process that spans through the entire software life cycle even though some phases may not produce a document (Harmon and Metz, 2014). The kinds of documents that may be produced include:

i. Process documents (plans, reports)
ii. Design documents (e.g., Functional Specification Document (FSD), Software Design Document (SDD), Interface Design Document (IDD))
iii. Release documents (e.g., user manuals, training notes, technical manuals, compliance document)

The software expert may require evaluating the consistency of the document content. He also needs to ascertain the level of compliance the document has with evaluation criteria by applying more than one evaluator and integrating their anomaly observations afterwards. The expert may make estimations on the evaluation result uncertainties by making use of the different observations submitted by the evaluators. The results are essential in building standards for comparison using historical data as the source of experience.

Evaluation of source code is directly dependent on the quality of products built from that source code on one hand, and controlling and estimating evaluation uncertainties on the other hand. Evaluation of source code involves intensive manual study of the code alongside automated analysis (Harmon and Metz,

2014). This gives minimal uncertainties of the results from both approaches. The first activity of source code evaluation is to sample the code that was submitted for cross-examination. The source code receives different reports depending on the sampling technique used. This in turn, produce different uncertainty results. The use of several evaluators gives opportunity of reducing evaluation uncertainties and also raises the potentiality of having accurate estimation of uncertainties. The results of manual evaluation can be likened to one or more automated code analysers. The output however, must be scrutinized manually with the intention of removing output noise as well as evaluation uncertainties. The results from source code evaluation is usually stored in a historical database which permits the development of comparison standards built on statistical basis.

Software executable evaluation, also known as software testing, implements testing processes to mainly ascertain progress of either design or performance (MITRE Corporation, 2021). The software testing/evaluation criteria is normally defined by the requirements model. Modern control on software testing (e.g., automation, design of experiments, test environment characterization) determines the best techniques that will suit the given software's testing process:

i. Designing test cases using some techniques that enhances the design of experiments which optimizes coverage (and not for searching of defect);

ii. Building test scripts and datasets from designed test cases using automation principles;
iii. Executing the test scripts by applying test automation; and
iv. Plotting a graph of the requirement to estimate test coverage and using the estimates to determine results uncertainty.

This testing methodology is a sustainable means for achieving a reduction in manual testing for the purpose of increasing testing coverage, iteration and result credibility. Software testing is normally seen as the key approach for the evaluation of software quality and determination of software quality assurance to build reliance on software workability and functionality. The criticality of software testing draws the software expert's special attention to managing this activity. The effects of the inadequacy of required tools for software testing in the software industry was summarized by Farooq and Dumke (2008) thus:

i. higher number of failures indicating poor quality
ii. higher cost of software development
iii. delayed release of software to the market because of testing inefficiencies
iv. higher cost of market transaction

In an attempt to manage testing, two questions come to mind:

i. When should testing be stopped and software be released?
ii. Is the testing being carried out in an effective and efficient way?

Farooq and Dumke (2008) are of the view that monitoring and putting these activities under control can be achieved by employing a continuous "in-process" and "post-process" evaluating approach. The tracking of the evaluation progress may involve determining the performance of the employed techniques, the efficiency of the activities that were performed and the tools utilized for testing.

Process evaluation involves comparison of planned and actual processes against best practices. The software designer collects observed information about the process from requirement documents (including project plans, and product document). He also gathers survey outcomes from the persons that conducted the survey exercise, and also gets the results of the process observation directly from independent evaluators. He uses the collected data to construct models of planned and actual processes. He further analyses the obtainable practices within the industry in comparison with the standards of the process being evaluated to arrive at a guide for constructing a process reference model. He then makes a comparison between the real process and the estimated process and also compares them with the process reference model where their differences result in anomaly observations.

Harmon and Metz (2014) designed the following format that an evaluation report can take:

i. Evaluation objectives and scope

a) Objective of Evaluation

b) Scope of Evaluation

c) Assumptions made in the cause of evaluation

ii. Method of gathering data and grouping of gathered data

a) Approach to sampling of product

b) Approach to manual evaluation

c) Approach to evaluation automation

iii. Method of analysing data and analysis results

a) Analysing data manually

b) Method of data filtering automation

c) Manual and automated data combination process

d) Approach to estimating the risks in usage

iv. Clarity issues in the evaluation process and their consequences

v. Conclusions, bottlenecks and future precautions

vi. References

Farooq and Dumke (2008) ascertains that as far as the quality of software is concerned, evaluation remains a key tool to its assurance. A program meant to tackle software quality issues should involve the following:

i. establishment, implementation, and control of requirements,

ii. establishment and control of methodology and procedures, and

iii. software quality evaluation.

A module the evaluates the quality of software is intended to assess the activities, processes and methodologies of software development.

2.1.2.2.1 Software Testing in Software Evaluation

Software testing is the primary and commonest validation technique used in evaluating the functional quality of a software product (Lysne, 2018; Chappell, 2018). Software testing is a complicated activity among other software development activities. A software testing domain comprises a number of elements including the methods and techniques of testing, the tools employed, the set of standards to be followed, the required measurements to be made, and the experimental knowledge, among others as shown in Figure 2.4.

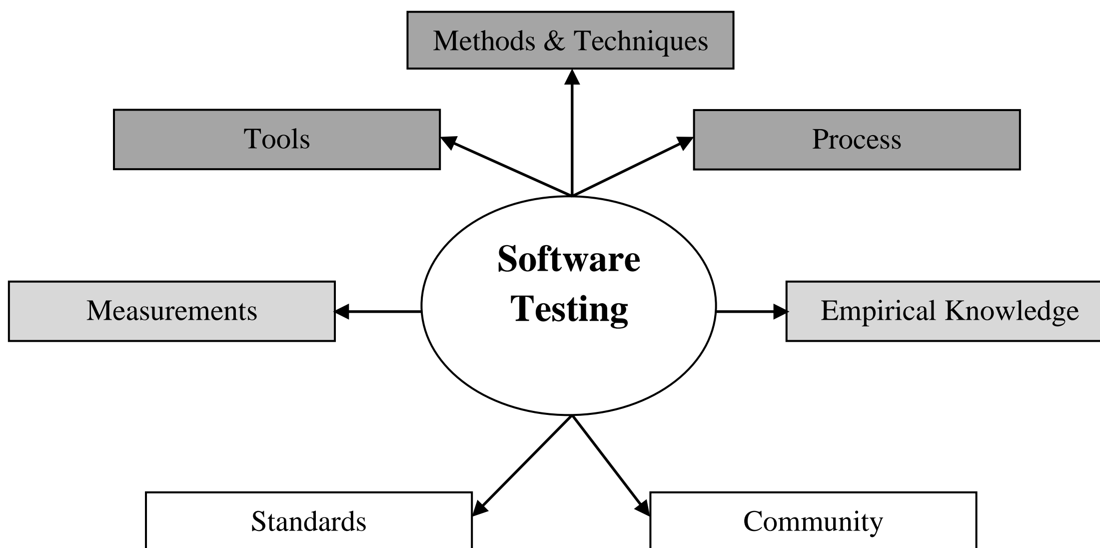


Figure 2.4: Major Software Testing Elements

(Farooq and Dumke, 2008).

Software testing is a software engineering niche that is largely influenced by the zeal to producing software with high quality. The efficiency of software testing

and the period of quality software release is given by the effective evaluation of the activities involved in the testing process. In other words, the activities associated with software testing are viewed as important phase in software development cycle. However, the process of testing is cost intensive and involves adequate development effort and time. Despite the much-accorded importance to software testing in software engineering research, it is yet unable to yield substantive methodologies that are capable of finding every fault in a software product. This is accountable to software complexities, test oracle problem (i.e. automatic recognition of a defect behaviour), and the situation of too large and too complex test cases (Lysne, 2018). Therefore, an important aspect of the software product has to be chosen for testing using some criteria that would reveal the characteristics that aggregately give the quality of the software product.

Altexsoft (2019) identified four types of software testing based on the testing objective to include Functional testing, Non-functional testing, Structural testing and Confirmatory testing. These software testing types are considered as the major among others because there are myriads of others even though they are usually misunderstood. Functional testing refers to black-box testing usually done when only the running application of a software is available (Lysne, 2018). It tests the software components or system to ascertain that the requirements are met or to establish the fact that business processes are

effectively implemented. These are observed through the component or system performance and ease of use. Cingil and Sozer (2022) implemented black-box testing on embedded systems which operated on the basis that only test cases related to error-prone files are selected for testing. Non-functional testing entails the unleashing of the quality characteristics of the software component or system. Structural testing implies white-box testing which requires that the internal structure of the software component or system must be known. Finally, confirmatory testing or re-testing refers to a situation in which a software update or modification is tested to ensure consistency in system functionality. Hence, it can be said that software testing plays the role of a technique for evaluating development artefacts and also an instrument for evaluating software quality.

2.1.3 Software Intelligence

The concern about the efficacy of software performance together with its precision to exactly perform as intended gave rise to the innovative idea of software intelligence. Software intelligence came into play to solve the problems associated with development process which eventually lead to poor performance. This makes it important that software intelligence is used to express the specifications that covers the domains of error, performance, deployment and usability. This will help in the check and sustenance of software health, bearing in mind that relying on user feedback may not yield adequate information about the software performance (Harley *et al.*, 2017).

Software intelligence obviously is concerned with the issues of software reliability and quality. Hence, it probes users' activities on software products and maps out strategies of solving pending problems they encounter. By so doing, it provides an in-depth idea about software-user interaction as a way of showcasing software performance towards achieving user satisfaction. It further gives an optimized decision on how defects can be fixed with minimal human efforts. As it were, software intelligence has some inclinations on business intelligence (Hassan *et al*., 2010). This is because the quest for finding concepts and techniques that are capable of sustaining fact-based decision making in business gave rise to the idea of software intelligence. Heartening enough, software intelligence has yielded far more beyond its initial functions.

2.2 Theoretical Literature

2.2.1 Software Evaluation Criteria

The term "criteria" can be described as the part of a design or evaluation attribute which can be measured. These measurable aspects are given as standards which ensure that a satisfactory implementation of both technical and operational characteristics can be measured (AcqNotes, 2020). Despite the seemingly clarity of the definition, Gediga *et al*. (2015) are of the view that software evaluation is hitherto scantly addressed by available literature as regards the specification of design principles and the criteria for evaluation that are generic in nature.

In order to achieve effective criteria for evaluation, certain factors have to be identified vis-à-vis the attributes to be evaluated (AcqNotes, 2020). Most times, evaluators make use of usability attribute to specify the criteria for evaluation owing to the universality of usability in describing all software systems. The international standard ISO 9241 – Part 11 (2018), the foundation to the methodologies associated with the standards for human computer interaction (HCI) designated as "Ergonomic requirements for office work with visual display terminals" (ISO 9241-11, 2018), emphasizes that the efficacy and satisfaction derived from making use of a software product to realize some specific goals is termed usability. Usability can therefore be described as having the features of effectiveness, efficiency and satisfaction. These features do not translate to measurable criteria, but only a foundational strategy towards meeting the criteria (ISO 9241-11, 2018). Evaluation involves adequate "operationalization" of the system documents and standards which determines the implementation approach adopted by the system. ISO 9241 – Part 11 (2018) gives a clear description for the usage of the approach covering the aspects of the user, the required tasks, the tools to be used and the operational environment.

Based on the definitions of the components that describe the mode and scope of usage, Gediga, *et al.* (2015) opined that an evaluation task is supposed to consider the following restricting factors:

i. The category of users with respect to experience, age, gender and/or other characterizing attributes.
ii. The various forms of operations that are performed by the users.
iii. The study environment which includes controlled laboratory conditions, unstructured field survey, etc.
iv. The type of object to be evaluated such as a software increment, a system model, a trial version, or a fully installed system.

As a matter of fact, the above-mentioned restrictions are recommended for all forms of evaluation including expert-based evaluation to ensure clarity and validity of evaluation outcomes. Gediga *et al.* (2015) identified four derivative (though not limited to four) techniques for the creation of evaluation criteria:

i. The partitioning of the evaluating principles based on order of preference using some sort of expertise until there is a clear means of measuring the leaves with a well-defined procedure. This technique is termed "top down I".
ii. The partitioning of the evaluating principles based on order of preference using some sort of expertise until the leaves become indivisible. This technique is known as "top down II" in which the operations of the technique are driven by some tailored procedures.

iii. The correlation of classified methods and some corresponding criteria or a practical mapping of item with criteria using assignment methods. This is called “bottom up” technique.

iv. The “operationalization” of evaluating principles, to embrace the required attributes to be measured as well as the quality of the evaluated objects.

Harmon and Metz (2014) are of the view that the software product’s evaluation for quality is largely dependent on the genuineness and/or acceptance of the evaluation criteria to the users of those products. They opined that the much-needed evaluation criteria should not fail to emanate from authoritative sources. This singular act minimizes the dependability of the evaluation process as well as the corresponding uncertainty owing to the effect of subjectivity. Harmon and Metz presented a table (shown in Table 2.1) of the different types of software products and the sources of their evaluation criteria:

Table 2.1: Sources of Evaluation Criteria for Software Products (Harmon and Metz, 2014).

SOFTWARE PRODUCT TYPE	EVALUATION CRITERIA SOURCES
Process documentation	Internal consistency, industry standards (e.g., IEEE, ISO, NIST, ITIL, PMI, organizational), industry best practices described in technical literature
Design & release documentation	Internal consistency, functional requirements (e.g., BRD, RSD, use cases, user stories), industry standards (e.g., IEEE, ISO, NIST)

Source code	Industry & organization coding standards & guidance (e.g., secure & reliable computing communities)
Executable code	Functional & technical requirements (e.g., BRD, RSD, use cases, user stories, requirements databases, applicable legislation), specialty standards (e.g. NIST SPs)

2.2.2 Classification of Evaluation Models

Evaluation models are responsible for determining the set-up for evaluation. Gediga *et al.* (2015) are of the view that evaluation set-up consists of

i. Making the right choice of techniques suitable for the evaluation life cycle, and
ii. Using the software objects and the criteria for measurement as the basic tools for achieving effective evaluation.

Evaluation models could make provision for establishing efficient evaluation procedures whose activities may promote provide standardization of action policies. They also play the role of comparative tools that differentiate various types of software evaluation. In general, the activities of all descriptive evaluation techniques are usually engaged in mutual operation with some kind of predictive procedure in order to realize an evaluation model that can be applicable.

Deploying evaluation models is seen as a good practice which supports the management of software quality. This is in agreement with ISO/IEC 25023

(2016) which describes an evaluation model as a sequence of properties and the corresponding interactions that exist among them which lays the foundation for the specification of quality requirements and evaluation.

Evaluation models are classified into three:

i. Method Driven Models:

This class of models provides evaluation set-up based on a collection of techniques. In a broader sense, the ideology of the models of this category is centred at the operation style of evaluation techniques and the regulations that guide evaluation tasks both at the preparatory and advanced stages. Method driven evaluation models are not as simple as they appear. The models are considered suitable if the evaluation process can effectively handle the user and designer problems.

ii. Criteria Driven Models:

This class of models defines and refines relatively abstract criteria. This means that the evaluation done with these models intends to measure the criteria. Criteria driven evaluation models make some assumptions about the design structure whose executions define evaluation criteria and also give instructions on how the criteria can be used to derive “measurables”. Because This class of models pay much attention to evaluation criteria and measurables, hence there is

no close connection between the evaluation model and a design model e.g. ISO 9241 (2018) and ISO/IEC 25010 (2011).

iii. Usability Engineering:

This is an evaluation model that came into existence due to the needs of a specific life cycle model. Usability engineering involves the rigorous bur consistent fusion of different methods and techniques contained in the system development process whose combined activities are needed for developing usable software. It is further described as a process which defines and measures the usability of software.

2.2.3 Disadvantages of Conventional Software Evaluation Models

In line with the observations of Chen *et al.*, (2011), the conventional software evaluation models generally have several limitations despite being widely used.

i. Conventional software evaluations are usually done manually which of course, is a time-consuming process.

ii. Software evaluation, especially when carried out by external experts suffer from management constraints in the sense that, management may withhold certain artefacts from being accessed by the third parties for some security reasons.

iii. Due to human sentiments, conventional evaluations are often subjective. Hence, the result is always biased.

iv. The conventional approaches highly require experts for qualitative formulation of questions and proper evaluation.

2.3 Empirical Literature

2.3.1 Software Evaluation Metrics

Software can neither be tangibly felt, either by seeing or touching but it is vital for effective use of computer systems. Hence the measurement and evaluation of software quality attributes is necessary. This is achievable by identifying and applying metrics to software products that promote and assess their quality. These quality metrics are essential for software quality evaluation.

A couple of metrics are formulas which are presented as functions (i.e. calculable), whereas a reasonable number of them are represented as basic countable values. Metrics are said to be countable if they originate from raw data of various sources including log files, video observations, interviews, or surveys. Countable metrics may be in the form of progress level of a project measure in percentage, success to failure ratio of the project, the rate at which a help menu is used, the time interval between error detection and error correction, the quantity of elements built into a user interface, etc.

Others examples of countable metrics include:

i. The comparative analysis of sales using previous and present data to ascertain if there is a boost in sales after software improvements.

ii. Duration of searching a database for an information.

iii. Test scores and survey feedbacks in connection with a virtual training program.

iv. Count of errors, critical errors, as well as successes in performing an activity.

v. Development and other logistics costs when erroneous software is to be rewritten.

Calculable (refined) metrics, on the other hand, are the results of arithmetic manipulations, program sequence, or investigations that are done on the basis of observed raw data or countable metrics. As an example, task effectiveness (TE) is a calculable metric. Task effectiveness can be calculated using a formula proposed by Bevan and Macleod in mid-1990's and is given thus:

$$TE = \frac{\text{Quantity} * \text{Quality}}{100} \tag{2.1}$$

2.3.1.1 Reliability Metrics

The probability of no failure occurring over a period of time during which a software is executed in a specified environment is termed software reliability (Kaur and Bahl, 2014). Software reliability, therefore, has much to do with the maturity, fault tolerance and recoverability of the software in discourse. Based on this note, there is need for the establishment of a well-drawn objectives that reflect the quality expected by the user. Also, it is necessary to determine a set

of supplementary objectives which will propel the realization of the user quality goal (IEEE Std 14764, 2006).

Software reliability is yet unclearly described to identify all aspects of software that have inclination to reliability. This is another way of saying that there is no best measurement procedure to software reliability and its associated aspects. However, there exist reliability metrics that serve as units for measuring the reliability of software systems. The measurement of the reliability of a software system is done by taking census of system errors and relating them to the service-requests placed on the software when the error status has not been exited.

The following are some of the metrics used in measuring the reliability of the functions of software systems:

i. **Rate of Occurrence of Failure (ROCOF):** ROCOF is a software measure that takes note of the number of times in which a system experiences failure. It describes the rate at which errors occur unexpectedly (Kaur *et al.,* 2014). ROCOF of a software product can be measured by executing and observing the way a software product operates over a period of time. ROCOF value can be calculated as:

$$\text{ROCOF} = \frac{\text{total number of failures observed}}{\text{duration of observation}} \tag{2.2}$$

ii. **Mean Time to Failure (MTTF):** MTTF is the mean of the time instants at which errors occur successively, i.e., the time interval between consecutive failures (Kaur *et al.,* 2014). MTTF is measured via the recording of n-failure outcomes. Let the failures occur at If the time instants t_i, t_{i+1}, t_2,.....,t_n are the times at which failures occur, then MTTF can be calculated as:

$$\sum_{i=1}^{n_e} \frac{t_{i+1} - t_i}{n_e - 1} \tag{2.3}$$

iii. **Mean Time to Repair (MTTR):** The occurrence of failure in software operation demands for a corrective measure to be done over a space of time to fix the error. MTTR is a software measures that takes note of the total number of times a system goes down and the period of time it takes to detect and fix the error that caused the failure (Kaur *et al.,* 2014). It is calculated as:

$$\text{MTTR} = \frac{Tot\ downtime}{noutages} \tag{2.4}$$

iv. **Mean Time Between Failure (MTBF):** MTBF is a software measure whose outcome depends on the combination of the values of the MTTF and MTTR (Kaur *et al.,* 2014). It is given thus:

$$\text{MTBF} = \text{MTTF} + \text{MTTR} \tag{2.5}$$

v. **Probability Of Failure On Demand (POFOD):** POFOD gives the possibility of a failure occurring when a system is being requested to perform a task (Kaur *et al.,* 2014). It is given by the number of failures encountered by a system over a number of inputs.

vi. **Availability:** Software availability is given by the possibility of a system being available for a specific time frame for usage (Kaur *et al.,* 2014).

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} * 100\% \tag{2.6}$$

vii. **Reliability:** As a unified software metric, reliability is given by:

$$\text{Reliability} = \frac{\text{MTBF}}{(1 + \text{MTBF})} \tag{2.7}$$

2.3.1.2 Usability Metrics

Software usability metrics can be defined as software measures which gives numeric assessment of the impacts made by software usability on current users, potential customers, and other groups that may be interested in the software. They are essential in order to ascertain easy understanding, learnability and operability of software. When analysing the progress-report received from a usability project, there is usually a scrutiny of “what,” “when,” “where,” and “how” metrics will be used and the foundational properties that will prepare grounds for the incorporation of the metrics. This means that the following questions have to be considered: What are the factors that would be measured?

When will the measurement take place? Where will be the best platform for the measurement? How would the measurement be performed? The answers to these fundamental questions are derived from the opinions of the software stakeholders in relation to their specified requirements and verified specifications. For instance, a request for quicker operation of software would translate to the use of execution time as the metric. Also, a request for the display of emotional features on user interface would mean an inclusion of those emotion-tags to measure them. More so, a request for a reminder of training that were done virtually would mean that a user-recall has to be measured. At the stage of determining metrics, the methodology is essentially scrutinized to understand the conforming tests and experiments as well as the rudimentary studies to be made in order to arrive at the needed solution to the needs of all stakeholders. The evaluator is also expected to understudy the potential outcomes of the subjective and objective implementation of metrics, the principles that guide good testing and experimentation, as well as the sources of relevant metrics (such as field operation, laboratory experiments, tracking systems). It is also essential to carry out a review of the factors and features that reveal usability. Usability of software is an important attribute that gives satisfaction to users over a safe usage of the software and an increased completion and accuracy of tasks performed with the use of the software. (Seffah *et al.*, 2006).

A school of thought is of the view that the application of metrics to software usability is done due to the following (and other) reasons:

i. Providing better understanding to stakeholders on the need for software usability,
ii. Providing potential customers with expected return on investment calculated for usability purposes,
iii. Providing visual representation of usability findings (including graphs, pie charts, and bar charts), and
iv. Providing stakeholders with the outcomes of tests, experiments and consultations that are usability inclined.

Software usability can be measured at software component level. Bertoa and Vallecillo (2006) identified three measurable concepts that function in this regard. The three concepts are

i. Documentation quality,
ii. Problem complexity and
iii. Solution (or Design) complexity.

2.3.1.3 Efficiency Metrics

Most of the available efficiency/performance metrics have gotten descriptions by several individual experts and consequently subjected to tests in a few environments. (Malathi and Sridhar, 2012) Sequel to this, the metrics are not very satisfying. This, as identified by Malathi and Sridhar (2012), is due to lack

of two basic characteristics whose effect can be felt singly or collectively namely:

a) Competent theoretical concepts

b) Validation of experiments which have statistical significant.

Efficiency metrics try to expose time and resources behaviour inherent in a software. Some of the metrics are:

i. **Mean Magnitude Relative Error (MMRE):** This serves the purpose of assessing the prediction of software performance. MMRE has the advantage of comparing software performance over several datasets. Also, the unit of measurement, whether in work-per-hour or work-per-month, does not affect software performance prediction. Mean Magnitude Relative Error is given by

$$\text{MMRE}(\%) = \frac{1}{n}\sum_{i=1}^{n}\text{MRE} * 100 \tag{2.8}$$

$$\textit{where}\ \text{MRE} = \frac{|\bar{\text{E}} - \text{E}|}{\bar{\text{E}}}$$

n = number of projects

$\bar{\text{E}}$ = actual effort

E = estimated effort

ii. **Prediction (n):** The percentage at which projects contain errors at absolute relative condition less than a fixed value *n* gives the prediction of software at level n.

iii. **Mean Absolute Relative Error (MARE):** The formula for Mean Absolute Error is given by

$$\text{MARE}(\%) = \frac{1}{n}\sum_{i=1}^{n}\text{MRE} * 100 \tag{2.9}$$

where $\text{MRE} = \left|\frac{\bar{\text{E}} - \text{E}}{\bar{\text{E}}}\right|$

iv. **Balance Relative Error (BRE):** This is given by

$$\text{BRE} = \frac{\left|\bar{\text{E}} - \text{E}\right|}{\min(\bar{\text{E}}, \text{E})} \tag{2.10}$$

v. **Median Magnitude of Relative Error (MdMRE):** MdMRE is given by

$$\text{MdMRE} = \text{median}(\text{MRE}) \tag{2.11}$$

vi. **Root Mean Square Error (RMSE):** Root Mean Square Error is a metric that measures the variations that occur in model's prediction or estimation against the actual values gotten from the real entity that was modelled. RSME usually takes a repeated approach in order to give a mean value. It gets its value by squaring the root of the mean square error (which is a measure of the average error magnitude. The formula is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{\text{E}} - \text{E})^2} \tag{2.12}$$

2.3.1.4 Functionality Metrics

Functionality metrics are units of measure for accuracy, adequacy, interoperability and compliance of software to its intended functions.

A. J. Albrecht developed the Function Points (FPs) in 1977 as a metric based on the functionality of the software delivered by an application as a normalization value. Function Points try to quantify the functionality expected of a system, i.e., what the system performs. Function Point is not a single characteristic; it is a combination of several software features:

i. Number of inputs that come from the external
ii. Number of outputs that come from the external
iii. Number of inquiries made from the external
iv. Number of files that are stored within the internal
v. Number of interfaces that communicate with the external

Function point of an application is found out by counting the number and types of features/functions used in the application. It is given by the following equation:

$$\text{FP} = \text{Count-total} * \text{CAF} \tag{2.13}$$

where $\text{CAF} = \left[0.65 + 0.01 * \sum (f_i)\right]$

$i = 1$ to 14

fi = 14 questionnaires showing the complexity adjustment value/factor-CAF. Usually, the value of $\sum f_i$ is provided.

2.3.1.5 Maintainability Metrics

The IEEE Standard Glossary of Software Engineering Terminology defines maintainability as the ability to modify a software system or component with less effort for the purpose of correcting faults, improving performance or other attributes, or adapting to updated or innovated environment. More concisely, maintainability refers to the probability of carrying out a maintenance activity within a stipulated interval of time which ranges from 0 to 1.

The following are some of the metrics used in measuring the maintainability of software systems:

i. **Technical Debt (TD):** Technical debt is the total amount of effort in hours required to reimburse all debts in the project.

ii. **Lines of Code (LOC) (or Source Line of Code, SLOC):** LOC is the number of lines of code (excluding comments and white spaces) in subroutines, files, etc. The commonest way of measuring LOC is to count the lines that contain executable program statements and ignoring the ones that contain comment and whitespaces (Viljanen, 2015).

iii. **Technical Debt Density (TDD):** Technical debt density is the ratio of TD to LOC.

$$TDD = \frac{TD}{LOC} \tag{2.14}$$

iv. **Maintainability Index without Comments (MIwtC):** An index value excluding code comments that gives the idea of how easy the code can be maintained.

v. **Maintainability Index with Comments (MIwC):** An index value including code comments that gives the idea of how easy the code can be maintained.

vi. **Maintainability Index (MI):** Maintainability index is given by the sum of MIwtC and MIwC, i.e.

$$MI = MIwtC + MIwC \tag{2.15}$$

vii. **Halstead Volume (HV):** These are Halstead complexity measures which include volume of the program, poor understandability and coding effort calculated from the number of distinct and total operators and operands.

viii. **Code Complexity (CC):** (McCabe's cyclomatic complexity) is the number of execution paths that exist in an entire program code. It has functional inclination to testability. This is because each execution path is expected to be verified with a suitable test case.

ix. **Unit test Coverage:** Unit test coverage is concerned with the identification of any code segment that are subjected to automated unit testing. Unit test coverage gives the idea of the magnitude of code segments that run during the unit test execution. That is why it is able to measure the code segments that are covered by tests.

x. **Maintainability:** Maintainability is a function which is involved with directly measurable attributes A_1 through A_n.

$$M = f(A_1, A_2, \ldots, A_n) \tag{2.16}$$

2.3.1.6 Portability Metrics

Software quality views portability as a basic concern. Software portability is the ability of a software to be moved from one runtime platform to the other without demanding for some modifications or full rewriting before it runs (Lenhard and Wirtz, 2015). The metrics that measure portability of a software basically evaluate the costs of transferring software from one platform to another and the costs of rewriting the same software codes from scratch and establish their relationship in a quantified expression. Software portability is bound to the times it takes the software program to run. A program element is said to be portable if majority or all of runtimes are in support of its execution. That is why the measurement of software portability takes the runtimes of process executions into account, devoid of theoretical consideration of the problem only.

The following are some types of metrics used in measuring the portability of software systems:

i. Basic portability metric: Basic portability metric is a measure of the relationship between the number of elements that are problematic and the total number of elements.
ii. Weighted elements portability metric: Weighted elements portability metric extends basic portability metrics by considering the degree of an element.
iii. Activity portability metric: Activity portability metric is similar to weighted elements portability metric. The difference is that its consideration focuses on the software activities rather than elements.
iv. Service communication portability metric: Service communication portability metric is also similar to weighted elements portability metric. However, this form of metric is meant to measure only the activities that communicate services and not the elements.

In general, the algorithm towards the measure of software portability involves the following steps:

i. Record total number of ports (e.g., browser and version, operating system and version, programming language, processor make and

speed, software modules (units) etc.) available within the environment.

ii. Identify the total number of successful ports within the environment to measure the portability.

iii. Apply the metric to measure the portability.

iv. Repeat steps i to iii for a different application software.

v. Repeat steps i to iii for a different application software in a different environment.

vi. End.

$$\text{Portability} = \frac{\text{Number of successful ports} * 100}{\text{Total number of ports}} \quad (2.17)$$

2.3.2 Existing Software Evaluation Models

Numerous research works have been carried out in the area of software evaluation and ultimately, the estimation of the quality of software systems. This section abridges some of the significant works made by researchers in the area of software evaluation and software quality estimation:

Some foundational researches carried out in the last two decades assessed quality in software product in the context of "safety-criticality" and "mission-criticality" of the software product and in the business it is utilized. They

analysed the views of software quality and outlined a mixture of product attributes that contribute to user satisfaction. The works gave a consensus description of software quality measurement as an aid to establishing baselines, predicting likely quality, and monitoring improvement.

Some other studies found in literature that took place within this period took the direction of analysing the established techniques for software estimation and measurement such as Boehm's COCOMO (Constructive Cost Model) and Albrecht's FPM (Function Point Method). These analyses were based on their comparison with the then new software development strategies (such as object-orientation, framework/component-based development and incremental prototyping) which helped in ascertaining the viability of their parts and the usefulness of the entire techniques with respect to available technologies. The results of these analyses motivated the researchers to develop new solutions which had much inclinations to software estimation, productivity analysis and software quality assessment.

In the early 21st century, many researches in the area of software quality concentrated in the niche of physical evaluation of candidate products, that is, administering actual tests on software products. Scholars like Dean and Vigder presented some evaluation techniques used to select Commercial Off-the-Shelf (COTS) software components for systems development in the year 2000. Their

study made a proposal of a selection-based evaluation system meant to provide support that would make a suitable selection of COTS products. The study provided advantages that centred around cost effectiveness and time management, having implications on precise component evaluation and implementation complexity reduction. The new approach combined the advantages of certain best processes observed from different methods with some newly introduced techniques. They study further identified black-box techniques as core strategies for software evaluations done in-context.

By 2006, the international standard ISO 9241-110 was already available. This served as a guide to researchers at that time into carrying out evaluations on software products. Motivated researches such as Hamborg and his group were noted for carrying out evaluation using usability questionnaire which they called IsoMetrics. The questionnaire was deployed to the measurement of the usability of a Hospital Information System, IS-H*MED. Their research proved the reliability of IsoMetrics since the technique successfully evaluated software applications that were useful to hospital information systems. Hence, the technique gave a boost to the screening of software usability in large organisations. The implementation of IsoMetrics questionnaire created room for identifying critical usability aspects as regards user-defined data types created by special users whose duty lines are concerned with performing the software's special tasks and functions. The results of the implementation showed low-level

ergonomic quality of the evaluated system. The work only supported usability of hospital management systems, rather than considering more software attributes.

Similarly, available literature also revealed that researchers delved into empirical investigations on ISO/IEC 9126-2 which was later withdrawn and replaced with ISO/IEC 25023 in 2016. The investigations were centred on determining whether categorization (as published in ISO/IEC 9126-2) can be proven correct and reliable in relation to users' satisfaction over their judgment on a packaged software product's quality. They investigated users' perceptions about product quality and discovered significant differences in their responses. These studies omitted the old version's reliability and compliance sub-characteristics (because of the difficulty the pre-testers had in relating those concepts). They mostly evaluated a single packaged-software product only. The study had a limitation of not examining other products to see if the dimensions of implementation are consistent across different software types.

There were other studies which presented evaluation schemes concerning the quality of software products via quantitative ranking. Thus, they presented schemes on how software products can undergo assessment and comparison for the purpose of defining the potentialities inherent in the software. Such works were not generic. Their attention was limited on only the methodologies that

focus on information systems and products which are commonly operated over public network systems and the internet.

In 2007, a model of generic measure and performance indicator which fused all essential aspects of the performance measurement process into a single but comprehensive framework was published. The work identified key concepts of performance measurement derived from a set of requirements that served as building blocks to the generic model. The identified set of requirements were used to model an architecture of generic nature meant to measure the performance of software systems.

Another study within the first decade of the 21st century investigated the strategies used in selecting software packages, the techniques used for software evaluation, the criteria for evaluation, and the supporting systems that aid in decision making towards evaluating software packages with the intent of providing innovations to software packages' evaluation and selection processes. Hence, the study proposed a generic methodology for selecting software packages and identifying evaluation criteria. The work was carried out by Jadha and Sonar (2009) and made the following findings:

i. Software packages have been evaluated over time using analytic hierarchy process,

ii. Software evaluation criteria of generic nature have not been enlisted and defined, and

iii. A model framework that addresses the methodology for software selection, techniques of evaluation, criteria for evaluation, and assistive systems for decision making is essentially needed to be developed.

The study only proposed a selection methodology and evaluation criteria and raised an opinion for the developing a generic framework that will perform the functions of software package selection, evaluation criteria identification, evaluation technique determination, and knowledge generation for assisting decision making over these activities. The study failed to develop the opinion.

Rana *et al.* (2010) identified two types of inconsistencies in the interpretations, representations and naming conventions of software product measures. They opined that software product measures are determining factors in creating and designing software quality prediction models. Rana *et al.* (2010) proposed a Unification and Categorization (UnC) framework which unified software product measures by applying frequency of use and usage history as criteria and also categorized software product measures with respect to three dimension namely usage frequency, software development paradigm, and software lifecycle phase.

Alvaro *et al.* (2010) considered the establishment of elements that make up software components for the purpose of certifying them as quality components. The study came up with a framework which considered four modules:

i. Determining the target features using modelled quality components,
ii. Determining appropriate methodologies for evaluating the model's provided features for the purpose of certifying the framework's technicality,
iii. Determining a process of certification which describes specific methodologies for evaluating software components in order to establish a standard for component certification, and
iv. Determining a definition framework that contains defined metrics required to evaluate identified features within a controlled environment.

The research work introduced four new sub-characteristics: Self-contained, Configurability, Scalability and Reusability but basically referred to quality component model.

Soliman *et al.* (2010) developed an automated tool for metric computation. Their work focused on the six metrics contained in the Chidamber and Kemerer metrics suite known as CK suite. Soliman and his team gathered the required information from class diagrams, activity diagrams, and sequence diagrams as well as the interaction that exist among methods resulting to more activity diagrams drawn with certain attributes. Using the gathered information, they

restricted the proposed automated tool to XMI standard file format in order to maintain an independent UML tool with some level of specificity.

ISO/IEC 25010 (2011) describes an evaluation framework that has two main parts namely:

i. the internal and external properties that refer to the quality attributes and
ii. the active properties that refer to the quality-in-use attributes.

Properties for internal quality make reference to the characteristics of the system which can undergo offline evaluation i.e. without executing the software, whereas the external counterparts refer to the other characteristics which must be assessed during the system execution. Both the internal and external properties can be harnessed by users during system operation and its maintenance state. On the other hand, the quality-in-use attributes are meant to make reference to the effectiveness of the product, productivity, security offered to the applications and satisfaction of users.

Isitan (2011) in his effort to ascertain the possibility of establishing a software quality model for free and open source software development, stated that a software product and the developed software can be called reliable if the end product can satisfy the requirements of metrics subject to criteria such as reliability, maintainability, and security as success indicators. He concentrated on the prospects of open source software quality by following a unified model

since he concluded that an open source software project can be seen to have failed when another project uses its code and advances the development of the software.

Jackson *et al.* (2011) carried out an evaluation that was based on certain criteria. The study presented diverse areas from which quality can be measured. The diverse areas are derivatives of ISO/IEC 25010 (2011) Software engineering — Product quality – Part 1. The measurement criteria are grouped as follows:

i. **Usability** (understandability, Documentation, Buildability, Installability, Learnability),
ii. **Sustainability** (Identity, Copyright, Licencing, Governance, Community) and
iii. **Maintainability** (Accessibility, Testability, Portability, Supportability, Analyzability, Changeability, Evolvability, Interoperability).

Their measurement involved thorough cross-examination of the software process and its resultant product to ascertain its conformity with specific characteristics or its ability to possess some underlying qualities required to be of a sustainable standard. The rate at which a software satisfies specific characteristics determines how sustainable it will be. The study restricted the evaluation to software sustainability without considering other software quality attributes.

Alrawashdeh *et al.* (2013) presented an enterprise resource planning (ERP) system-based quality model using the standards of ISO/IEC 25010 (2011). The quality model verifies conditional implementation of enterprise resource planning (ERP) systems in higher education institutions as either successful or failed. They suggested that the minimum quality characteristics required to create ERP quality model are six namely functionality, reliability, usability, efficiency, maintainability, and portability. These quality characteristics are measured using twenty-seven sub-characteristics. The study offered a comparative analysis of existing quality models and was able to identify the quality characteristics of ERP systems. However, it failed to identify the basic quality characteristics of the newly proposed quality model.

Akbari and Rajabi (2013) considered the functional and non-functional properties of Desktop free/open source Geospatial Information Systems (GIS) software while subjecting them to evaluation following three dimensional applications. They identified that Desktop free/open source Geospatial Information Systems (GIS) software have not gained enough popularity due to:

i. Free/open source software creates strategies that provides solutions to problems and gives user satisfaction less preference. This is contrary to patented software which tries to satisfy users using software services.

ii. Often free/open source software executions are dependent on command line not on a graphic user interface.

iii. Principally, in free/open source Geospatial Information Systems (GIS) software niche of research, there has not been any all-emcompassing type of software whose execution is done on both desktop and web platforms. Studies have shown that the non-existence is due to two reasons. To start with, the projects involved in free/open source software category usually start with definite goals and extend to scholar goals. On the other hand, it is a known fact that it incurs a huge cost to realize an all-encompassing software. Besides, free/open source software projects are usually unable to cope with the high extent of expenditure.

They evaluated free/open source GIS software with respect to three dimensional applications namely performance, scalability and usability. The work did not extend the evaluation of free/open source GIS software to accommodate more quality characteristics such as reliability, extendibility, testability, etc.

Misra *et al.* (2013) proposed a framework that aimed at measuring and validating software complexity. They framework designed the analysis to ascertain whether or not a particular software metric is qualified to be used for some measurements depending on the perspective of analysis. The study also considered the practical benefits of using the metric in cognisance of the theoretical and empirical validation factors such as measurement theory. The study further investigated how best the framework can be applied using cognitive functional size measure (CFS) which showed that the framework has

applicability in any form of software measure. The study also compared the framework with already existing ones and proved its betterment over others since it represented the required parameters for evaluation and validation of new complexity measure in a better format. However, the framework did not provide strong backing for the conditions and guidelines it provided for proper evaluation and validation of software metrics.

Lochmann (2013) presented a precise and assessable way for defining software quality. He proposed an explicit quality meta-model that describes the structure of quality models. Lochmann used the quality modelling approach to define product model of software systems and relied on the activity-based paradigm, which describes quality as the capability of software to support activities conducted with it. Based on the proposed quality meta-model, Lochmann formulated an approach for quality assessments which uses existing measures and analysis tools for quantifying the satisfaction of properties defined in a given quality model.

Bo Zhou *et al.* (2013) carried out a work on software quality evaluation and discovered its importance in validating software system's correctness. The correctness as contained in the study is ascertained via the following testing activities:

i. Test cases formulations and

ii. Test cases execution to validate the software system's behaviour.

Bo Zhou *et al.* (2013) further presented the quality of test cases to have direct effect on software quality since test cases are characterized by the ability to uncover hidden failures within the software product.

ISO/IEC 25000 (2014) is an update of the ISO/IEC 25010 (2011) framework. It is subdivided into 8 sub key features and characteristics – Functional Suitability, Reliability, Performance efficiency, Operability, Security, Compatibility, Maintainability, and Transferability. The framework constitutes a set of standards based on ISO/IEC 25010 (2011). Among other objectives, the framework is poised with the provision of guidelines in the development of software products in conformation with the specification and quality evaluation requirements. The framework perceived software security and compatibility as new characteristics which are responsible for grouping some former portability characteristics alongside those that were not logically part of the transfer from one environment to another. The framework further termed transferability as a portability extension. As with the ISO/IEC 25010 (2011), the framework followed a strict maintenance of the three different views in the study of the quality of a product.

Wang (2014) presented methods for evaluating a particular software quality attribute namely reliability, and indicated that stochastic modelling is of high

importance as far as software reliability is concerned. The importance came as a result of the ability of stochastic modelling to provide a unified framework which effectively analysed reliability from various aspects. The work was based on the assertion that the dependability software systems is enhanced via an effective investigation of their quality. The study by Wang (2014) adopted combinatorial methods, Markov models, software models as well as stochastic-based evaluation techniques.

Ciaschini *et al.* (2014) applied two predictive modelling techniques on some software products to determine the risks inherent in their individual operations. The predictive models were applied to the historical development of some European Middleware Initiative (EMI) packages. The study made this application for the purpose of identifying the risk factors of each package in comparison with their real history. The study also investigated how the models map the realization of the applications under evaluation. However, the study failed to identify predictive methods that provide better defect predictions.

Oberscheven (2014) collected and analysed the data for the software metrics to evaluate his software quality model and presented it to the people involved in the agile software development process. The results from the application and the user feedback suggested that the model enables a fair assessment of the software quality and that it can be used to support the continuous improvement of

software products. The model made use of a few metrics; hence it needs to be extended with additional metrics to accommodate additional aspects of software quality including the analysis of additional characteristics like Modularity and Reusability.

Figueiredo (2014) performed a research on evaluation and its prospects and identified evaluation as a control which emanates from the ability to scrutinize emerging methods, the techniques involved in the methods, the languages used to express the methodologies as well as the tools used in implementing the techniques involved in the methods. The study only inquired the evolution of evaluation but did not carry out some implementation(s) of evaluation to a particular problem.

Lyras *et al.* (2014) utilized Educational Data Mining (EDM) techniques to evaluate educational software designed to support learning. The study employed the techniques of prediction, feature selection and relationship mining in examining some data realized from experiments carried out at the Department of Education of the University of Patras with the intent of discovering some hidden rationale behind the utilized software's evaluation task. The study evaluated 15 educational software, considering a total of 177 evaluation criteria classified under nine educational software contexts: Instructional design, User

interface, Media and quality of information media, Aesthetics, Content, Navigation, Feedback and interaction, Usability and Ease of Use.

Sherief *et al.* (2014) advocated future popularity of the use of crowdsourcing in evaluating complex and sophisticated software systems. The study further foresaw the evaluation of highly variable software systems on the cloud since variable systems can easily work in diversified and unpredictable contexts. By definition, crowdsourcing involves online provision of solutions to problems by numerous online users who have the ability of providing non-expertise solutions to relatively simple tasks. This technique enriches and keeps the constant feedback of online users for knowledge updating which also help to arrive at a reasonable solution to problems in record time. Crowdsourcing also has the ability of introducing newer challenges especially those that have to do with proper organization of the online crowd and find a way of providing the right platforms that can accept and process their numerous inputs. The work by Sherief *et al.* (2014) focused on accepting/gathering the crowd's evaluation feedback. The study organized two focus groups to help in understanding the various aspects of the gathering and acceptance activity. The study therefore had the primary goal of maximizing software evaluation efficiency and scalability for the purpose of accommodating complex and highly variable systems in which the crowd's inputs are greatly needed. Unfortunately, the study did not analyse the focus groups thoroughly. It also failed to present a

proof that the results from online participants were confirmed or that the sample of users were large enough for the experiment. Besides, the basic requirements on how to engineer crowdsourcing evaluation, how to ensure correctness, and how to maximize quality using crowdsourcing is yet to be established.

Miguel *et al.* (2014) investigated the source of software quality whether to come from software process or the product itself. The study analysed software models and identified their strengths and deficiencies. The study concluded by adopting software product (precisely the final version) rather than software process as the provider of quality regardless of the fact that the duo has tremendous closeness and relationship.

Hlomani and Stacey (2014) analysed the state of the art in ontology evaluation, spanning through topics like the approaches to ontology evaluation, the metrics and measures used. They identified the two important aspects of ontologies: ontology quality and ontology correctness and attempted to make a distinction between the two as a way to better ontology evaluation approach. They also explored the notion of subjectivity as an important aspect of ontology evaluation. They paid particular attention to the influence of this subjectivity/bias on the overall measures of ontology correctness. However, they did not extend the notion of subjectivity to other aspects of ontology evaluation (i.e., ontology quality).

Harmon and Metz (2014) stated that actual software quality evaluation (SQE) vary from one situation to the other. Hence, there is need for practical solutions which ought to be fashioned and implemented in real time. The study represented the SQE process as an instrument (consisting of skilled evaluator and evaluation criteria that the evaluator is trained to use) observing the characteristics of a software product. The study also identified software products including documentation, source code and executable packages. The work identified different evaluator skills and evaluation criteria for each of these product types. The authors identified requirements modelling as an essential part of SQE. From their point of view, the final requirements model consists of two graphs:

i. Object graph – vertices represent objects & edges represent the interactions between objects and
ii. Dependency graph – a bipartite graph with vertices that represent both object properties and dependencies & edges that represent the mappings between the object properties and the dependency independent and dependent variables.

The requirements model defines the test criteria for software testing. This software testing strategy operates towards rendering manual testing insignificant in order to raise the potency of testing process ability to cover much area, iterate

and reproduce the sequence of steps when needed. The study opined that the sampling of the source code marks the beginning of software evaluation. Thereafter, all evaluators involved in the evaluation activity can utilize their expertise to ensure a decrease in the uncertainties that could be experienced from the evaluation results. This would be an assurance to increasing the accuracy of uncertainty estimates. Technically, they identified two basic variants of software product evaluation namely: Human-readable products and Machine-executable products. This approach requires more discipline than is usually applied in software quality assurance endeavours.

Gediga *et al.* (2015) based their work on the old ISO/IEC 9126-3 (now ISO/IEC 25023, (2016)) general framework for software evaluation. The work presented the three steps of evaluation (fixing the requirements, preparation of the evaluation and the evaluation step) as stated in the standard's old version in more detail. The findings of the work demonstrated that the techniques deployed in the study produced huge amount of overlapping results, an outcome of the combination of several evaluation techniques which needed to be evaluated as well.

Alashqar *et al.* (2015) performed a study dedicated to the development of evaluation framework which quantifies software quality via multi-criteria decision-making (MCDM) problems, initiated by some interactions among the inherent criteria. The study introduced what is known as *aggregator methods"

which included Arithmetic Mean (AM) and Weighted Arithmetic Mean (WAM). The aggregator methods were used by the authors to describe and compare the performance of MCDM with a fuzzy measure known as Choquet Integral (CI) approach. The study made the comparison using six quality attributes (namely Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability) as a guide to evaluating and ranking software alternatives. The study was validated using some real data gotten from case studies. The study failed to consider the evaluation of sub-criteria of quality attributes since there exists different software product views. It also did not use elaborate volume of data for the evaluation which is critical to determining reliability of the results.

Mansoor *et al.* (2015) presented a criteria-based approach whereby the goal model of goal-oriented requirements engineering is used to establish the functional goals as criteria for measurement. The study proposed a methodology used to measure the strength of relationships among quality goals and between functional goals. The functional goals are prioritized according to stakeholders' preferences. Triangular fuzzy numbers (TFN) and defuzzification process is used for prioritization, the developers input and risk tolerance is dealt by defuzzification of triangle fuzzy numbers (TFN). Thereafter, the process was applied to specified quality goals which are geared towards using quality models. On a final note, there was an evaluation of the dependencies among

quality goals and between functional goals. The study also applied the methodology to the ’cyclecomputer’ as a case study where 8 functional goals were established and stakeholders’ opinions were collected for these functional goals. They calculated the importance value of each functional goal and then integrated quality goals before prioritizing them according to their dependencies. The approach failed to integrate goal models and complete quality models.

Nakai *et al.* (2016) based their work on common standards, called the SQuaRE (Systems and software Quality Requirements and Evaluation) series provided by ISO/IEC. The study proposed a SQuaRE-based software quality evaluation framework. The proposed framework played a great role in standardizing product metrics and quality-in-use metrics originally contained and described in the SQuaRE series1. Precisely, there were 65 metrics deployed in the framework, 47 of which are quality metrics while 18 are quality-in-use metrics. The framework was designed to make use of documents, user tests and questionnaires to accomplish software measurement. With this configuration, the framework was found applicable to commercial software package/service product with the help of a case study. The major aim of the framework is to reduce ambiguities in metrics as well as clearly define inputs and outputs for quality metrics. The framework advocated for the standardization of product metrics, metric-based evaluation of the sufficiency of the quality (sub)-

characteristics, and identification of areas that need improvement using evaluation output. The framework obviously contained two parts namely (i) Product Quality and (ii) Quality-in-Use. The framework assigned product quality part with internal and external product quality characteristics and metrics based on ISO/IEC 25023:2016, whereas the quality-in-use was furnished with quality characteristics and metrics of quality in use based on ISO/IEC 25022:2016. The framework relied on manuals, specifications, design, source code, violations of the coding standard, test specifications, test results, and bug information in measuring product quality. They did not apply the framework to various domains to confirm its general applicability so as to improve feasibility and usefulness.

Munaiseche and Liando (2016) presented a research work that measured the usability of applying expert system through the use of tasks as interaction media. The study made use of questionnaire to gather data that helped to apply expert system in diagnosing skin disease in human. The questionnaire method measured usability by using tasks as interaction media. The study identified learnability, efficiency, memorability, errors, and satisfaction as usability aspects and became the first step that evaluated usability by applying expert system using task model. The study did not consider other quality attributes of usability, which include download time, navigability, interactivity, responsiveness, content quality.

To present a practical expert-based evaluation method for conformance testing with the international standard, ISO 9241 (2018) evaluator should be presented in detail to give an understandable integration into a comprehensive evaluation approach. The judgement of ISO 9241 (2018) is not user-based, and has poor reliability. The expert's judgement regarding evaluation items relevance and rating could constitute a bias on the final statement.

Kapoor *et al.* (2018) carried out an investigatory research on software fault prediction. The study presented an innovative strategy that guides the search for exact gaps to be filled to align industry requirement with research findings that drives the field of software fault prediction and its continuity.

Li *et al.* (2019) proposed a weighted network for software fault-proneness prediction. The study integrated developer contribution, module dependency, and developer collaboration relations in a Tri-Relation Network (TRN) to study their combined impact on software quality. The study employed network node centrality metrics derived from the network to predict fault-proneness. Li *et al.* (2019) posited that TRN can make provisions for a detailed and comprehensive insight into the manner in which developers interact with program modules rather than using networks on the basis of either a single or a paired relation.

Ayyıldız and Erkal (2019) carried out a study with the intent of finding a way to determine the number of bugs in developed open-source software projects.

Their work also considered the relationship between identified bug numbers and software quality metrics. At the core of the study, Ayyıldız and Erkal (2019) subjected about 20 open source game projects developed with java programming language to Static code analysis to measure software quality metrics. They also utilized linear regression to determine the relationship between software bugs and software quality metrics. At the end of the study, Ayyıldız and Erkal (2019) asserted that that it is possible to use software quality metrics to estimate the number of software bugs.

Omri *et al.* (2019) presented a combination of code complexity metrics, static analysis together with code churn metrics for predicting fault density in software systems. They furnished the predictor with the capability of building classifiers to distinguish fault-prone components from non-fault-prone ones inherent within the software. Omri and his group succeeded in obtaining early estimates of fault-proneness which is regarded as a boost to the efficiency and effectiveness of software quality assurance.

Uqaili and Ahsan (2019) proposed an approach that categorizes software bugs according to their severity. They also considered software bugs according to their priority basis and utilized the information to label software metrics' data. Uqaili and Ahsan further presented machine learning models for the prediction

of fault prone software modules, having used the labelled data to train the models via supervised learning approach.

Ghandorh *et al.* (2020) presented a systematic literature review, carried out to gather evidence on measuring software portability as a desirable attribute for their software quality. They focused on an overview of both used and proposed measurement metrics of software portability. They posited that different researchers have made efforts to understand measurement of software portability, however no census of these researcher was achieved.

Shafiq *et al.* (2020) analysed 227 articles in search of the extent in which machine learning can help reduce the effort/complexity of software engineering and improve the quality of resulting software systems. The study introduced a new machine learning for software engineering (MLSE) taxonomy following a systematic mapping study on applications of machine learning to software engineering. Shafiq *et al.* (2020) furnished MLSE taxonomy with the standard guidelines and principles of empirical software engineering which provided it with the capability of classifying the state-of-the-art machine learning techniques according to their applicability to various software engineering life cycle stages.

2.4 Related Work

This study has close relationship with previous works done in software process evaluation research. Particularly, we have drawn inspiration from some models that contain similar features with the intention of fine-tuning and introducing machine learning approach to overcome their limitations. One of these models (which is foundational to many other models) was proposed by Jeanrenaud and Romanazzi in 1994. They focused on the use of software evaluation metrics in a software product evaluation methodology called CDSEM (Checklist Driven Software Evaluation Methodology), designed by Software Quality Laboratory of Tecnopolis CSATA Novus Ortus. They considered software product as composed by different parts: software system, product documentation, user documentation, support services and distribution media. The methodology proposed the evaluation models in accordance with the standard ISO 9126 (Information technology - Software product evaluation - Quality characteristics and guidelines for their use) taking also into account the emerging new parts of the standard. The six characteristics defined in ISO 9126 (functionality, reliability, usability, maintainability, portability, efficiency), were exploded, for every component, into sub-layers of abstractions till to the identification of the measurable items (metrics). Moreover, the methodology identified, for each metric, tools and procedures for the evaluation. Jeanrenaud and Romanazzi also developed a tool on PC platform called CDSET (Checklist Driven Software

Evaluation Tool) to manage the methodology information base, results and reports. The tool aimed at guiding the user through the phases of the Methodology and allows him to select quality parameters and to suite the checklists according to the product to be evaluated.

In addition, this study has some similarities with Zhang's work done in 2002. Zhang (2002) took a look at the characteristics and applicability of some frequently utilized machine learning algorithms. Hence, made formulations of some software development tasks using learning algorithms. Zhang showed how machine learning algorithms can be used in tackling software engineering problems. He posited that machine learning should not only be viewed as a means of building tools for software development and maintenance tasks. Instead, it should be incorporated into software products to make them adaptive and self-configuring.

Also, this study is related to the work carried out by Zhang and Tsai in 2003. Zhang and Tsai carried out a general study on the application of machine learning in software engineering problems. They identified the characteristics and applicability of some frequently utilized machine learning algorithms. Zhang and Tsai used the identified characteristics and applicability as the basis for analysing the existing work in the niche area. Their work went further to

offer some guidelines on applying machine learning methods to software engineering tasks.

In a similar development, Brun and Ernst, (2004) carried out a research work which is related study to this study. The worked had its main focus on latent code errors. The authors applied machine learning in the study with the intent of finding latent code errors inherent in software products. They proposed a technique that reveal errors by applying system generated machine learning models to user-defined program properties to classify and rank properties according to their proneness to error.

In furtherance with the similarity to this study, Andrzejak and Silva (2008) presented a method for monitoring and modeling performance degradation caused by software aging. They investigated how machine learning (classification) algorithms could be used for detecting the so-called performance degradation. Based on the predictive power of these algorithms with several techniques, Andrzejak and Silva formulated a framework that aims at making the measurement-based aging models more adaptive and more robust against transient failures.

The model presented by Yahaya *et al.* (2011) in the work titled "Development of a Dynamic and Intelligent Software Quality Model" is also related to this study. Yahaya *et al.* (2011) integrated wrapper-based feature ranking technique

for building learning models and other methods to produce a complete algorithm for assessing software products using intelligent model. They utilized this technique to rank performance on the value of each attribute. Their work resulted in a dynamic intelligent model called i-PQF which is an extension of a previous quality model known as Pragmatic Quality Factor (PQF). i-PQF is capable of identifying and recommending to the environment if there is any new attribute to be included in the model. The i-PQF can be applied in investigating the development of quality model that is capable of noticing, learning and adapting to the changes in the environment and the corresponding information needs.

In addition, the work titled "Software Process Evaluation: A Machine Learning Approach" carried out by Chen *et al.* in 2011 have similar features with this study especially in the aspect of integrating machine learning with automated system. Chen *et al.* formulated a software process evaluation task as a sequence classification problem that can be resolved by machine learning techniques. Based on the proposed framework, the authors presented a new quantitative indicator, called process execution qualification rate, as an objective evaluation of the quality and performance of software process. The work employed supervised sequence classification to determine whether a process is qualified, i.e., is able to meet intended purpose. However, the work was limited to four software products and is not certain to be suitable for more complicated

software evaluation tasks. Chen *et al.* (2011) expected future works to extend and apply the framework to more complicated processes. Also, the work made some "precision" assumptions which may not hold in real-world situations.

This study is also related to the work done by Bindal in 2013. The work investigated how Markov Mode of software is drawn, its application and how it is used for estimating the quality of software. Bindal maintained that the main objective of software testing is to evaluate the performance of the software under given conditions without any type of corrective measure with known fixed procedures. Other objectives of software testing, as stated by Bindal, include:

i. To find perceptual structure of repeating failures.
ii. To find the number of failures occurring in specified amount of time.
iii. To find the mean life of the software.
iv. To know the main cause of failure.
v. After taking preventive actions checking the performance of different units of software.

This study is also related to the framework presented by Rana *et al.* in 2015 which applied machine learning approaches for prediction of software quality in large software organisations. Rana *et al.* (2015) employed machine learning in modelling the quality of given software and effectively used the collected

software metrics and historical data which are available within large software development organisations. They applied Pattern Recognition and Classification to predict a quality category in which a given software falls, during its development. The categorisation takes a top-down approach using some quantitative value as the measurement criteria.

The work done by a group of three researchers in 2016 with title "Machine Learning Strategy for Fault Classification Using Only Nominal Data" has close relationship with this study. In the work, Nicchiotti *et al.* (2016) suggested a strategy to use machine learning methods for fault classification purposes and diagnostics. They proposed a framework that implemented three different machine learning methods – Gaussian Mixture Model (GMM), Support Vector Machines (SVM) and Auto Associative Neural Networks (AANN). The framework considered some a priori knowledge about the faults to be classified in order to drive the behaviour of the machine learning methodology to be more or less reactive to the different faults. In a nutshell, Nicchiotti and his team presented a strategy for data driven fault classification when only healthy data is used for training.

Additionally, this work is closely related to Nwandu and Asagba's work in 2017. Nwandu and Asagba (2017) presented an automated model that uses different metrics to estimate the reliability of software systems. Their work was

based on software testing principles with the aim of maintaining one important aspect of software quality evaluation namely reliability - being able to measure and predict the system's reliability accurately. Their model estimates the reliability of given software systems using the context of execution speed. Nwandu and Asagba (2017) focused their attention on the failure times of given software and used the time instants to calculate some metrics as well as moving further to estimating the reliability value of given software which concentrated on the execution speed of the software. The model employed only a few metrics. It therefore requires that an extension be added as regards the metrics in order to accommodate additional aspects of software quality including the analysis of other software quality attributes.

In another study, Hammouri *et al.* (2018) evaluated three machine learning algorithms in their effort to predicting faults in software systems. The study created a model for this evaluation purposes which utilized historical data to predict potential faults in software systems. The three algorithms (Naïve Bayes (NB), Decision Tree (DT) and Artificial Neural Networks (ANNs)) are supervised forms of machine learning whose performance on evaluation process proved to be effective. This thesis however, employed reinforcement learning in establishing precise quality of a given software.

Using a combination of the best features of fuzzy logic and neural network, Pattnaik *et al.* (2018) presented a hybridized prediction model meant to measure a predictive quality of software. The predictive ability is a resultant effect of some specific features that are harnessed during the development process of the software. However, this thesis is in the job of providing a tangible measurement of software quality in numerical representation.

In furtherance of their research, Pattnaik *et al.* (2019) applied fuzzy logic in modelling a pragmatic framework that considers the contributory factors to the quality of software systems regarding the software attributes. The model simulated these quality factors with the view to quantifying them. In a way to move ahead of the work, this thesis assures to quantify the quality of software systems in numeric values.

Furthermore, Bajpai *et al.* (2019) carried out an investigative study to discover the academic performance of students using machine learning. The study applied machine learning algorithms to harness relevant details of each student from complex datasets. At the end of their study, Bajpai and his team asserted that machine learning is best suited for several real-world applications like analysing students' performance. The work used machine learning to determine student's academic performances whereas our model applies machine learning

to software evaluation with the intent to quantify its quality attributes in numerical terms.

This study is also related to that of Li *et al.* (2019). In the work, the authors proposed a revised benchmarking configuration for detecting faulty software code. The configuration considered several dimensions such as class distribution sampling, evaluation metrics, and testing procedures. The work asserts that predictive power is heavily influenced by the evaluation metrics and testing procedure. Also, classifier results depend on the software project. Hence, the study suggested that predictive accuracy is essential in predicting faulty software units. The work focused on the fault detection on software code, which is more or less, a sort of white-box testing whereas our model implements an intelligent black-box evaluation of software, no matter the type of software involved, to quantify its quality attributes. However, the fault detection of the work was predictive rather than being clearly measured via system performance.

This work further possesses similarities with that of Kim *et al.* (2020) titled "Unexpected Collision Avoidance Driving Strategy Using Deep Reinforcement Learning". Kim *et al.* (2020) generated intelligent self-driving policies in their quest to provide guidance on reward engineering in terms of the multiplicity of objective function. The study used the deep reinforcement learning to minimize

the injury severity in unexpected traffic signal violation scenarios at an intersection. Kim and his team designed two agents, one with a single-objective reward function and the other with a multi-objective reward function. Further, they used a deep deterministic approach to train self-driving agents in a simulated environment. The work applied reinforcement learning to achieve traffic control whereas our model applies the learning approach to software quality measurement.

In another study that has relationship with this study, Kopyltsov (2020) proposed an expert method for software assessment which is capable of evaluating the numerical values of software's practicality, integrity, efficiency, correctness, security, reliability, ease of use, evaluation, flexibility, the possibility of use in other conditions, mobility and the possibility of interaction. The author utilized quality metrics to arrive at a modified method for assessing software quality. The resultant modified method allows for both improving the quality of software and evaluating the material costs necessary to improve the quality of software. The work was centred around quality improvement and its cost effectiveness whereas the model presented by this study screens software to measure out its quality in numerical values vis-à-vis its functionality using quality attributes and metrics.

Similarly, Moreno *et al.* (2020) propose a flexible method to assess and improve the quality of requirements. The study applied machine learning techniques to emulate the implicit expert's quality function. The study further classified set of requirements according to their quality and extracted their quality metrics with a view to providing a procedure to suggest improvements in bad requirements. Moreno and his team finally automated process of inferring and applying the adapted quality rules. The focus of the work is on the quality of requirements instead of that of the system itself. Besides, there are no stated criteria for the assessment of the requirements quality.

Moreso, Kassie and Singh (2020) conducted a survey with university students on the user's perspective to improve the quality of a software, with the aim of ranking the important software quality factors. The study identified 10 most critical software quality factors that includes functionality, reliability, usability, efficiency, maintainability, portability, understandability, interoperability, operability, and aesthetic. The results of the survey motivated Kassie and Singh to propose a new user's perspective-based software quality model. The model qualifies software based on user response and ranks the quality factors also based on survey feedback.

Furthermore, Tsantekidis *et al.* (2020) proposed trading agent using Deep Reinforcement Learning approach to ensure that consistent rewards are

provided to the trading agent. The study designed a data pre-processing method that trains the agent on different forex currency pairs. Tsantekidis and his team introduced a novel price trailing-based reward shaping method which provided significant improvement on various performance metrics culminating in the enhancement of the overall performance of the agent. The agent presented in the work makes prediction on a traded asset whereas the interest of this study lies on the precise functional units that gives best outcome consistently, given the same test data.

With the idea that software quality prediction has been given attention by several researchers, Cowlessur *et al.* (2020) studied various machine learning techniques that have been applied to this study niche. The study looked at Artificial Neural Network (ANN), Bayesian Network (BN), Fuzzy Logic (FL), Decision Tree (DT), Support Vector Machine (SVM) and Case-Based Reasoning (CBR) alongside their related approaches. The result of the study insinuated that most machine learning-based prediction models aim to achieve quality prediction as early as possible during software development, the result expected to be of high efficiency in contrast with other techniques. However, the study opined that various vague areas are yet not explored in the business of applying machine learning in predicting software quality.

Similarly, Omri and Sinz (2021) conducted a survey with focus on the selection and prioritization of test cases as well as the early prediction of faults. The survey gave explanations why deep learning algorithms play conjugal roles in the dichotomy that exist among program semantics and system structure which is capable of predicting the presence of faults.

Furthermore, Chren *et al.* (2022) conducted quality evaluation on students' coding projects. The analysis was based on some metrics whose results showed the possibility of software quality course in quality improvement of student projects. The summary of existing works that are closely related to the proposed model presented in this study are summarized in Table 2.2.

Table 2.2: Summary of Most Related works.

AUTHOR	**WORK TITLE**	**WORK DONE**	**COMMENTS**
(Chen *et al.*, 2011)	Software Process Evaluation: A Machine Learning Approach	Presented a process execution qualification rate, as an objective evaluation of the quality and performance of software process.	The model was limited to only four software products and is not certain to be suitable for more complicated software evaluation tasks. It made some unrealistic "precision" assumptions.

(Bindal, 2013)	A Review of Markov Model for Estimating Software Reliability	Surveyed how Markov Model can be used to actualise reliability estimation with main objective of evaluating the performance of the software under given conditions.	The model only described how Markov model can be applied in failure detection using metrics but failed to implement the actual reliability assessment.
(Rana *et al.*, 2015)	Machine Learning Approach for Quality Assessment and prediction in Large Software Organisations	Applied Pattern recognition and Classification to predict a quality category in which a given software falls, during its development.	The model only focused on quality Categorisation using few metrics.
(Nakai *et al.*, 2016)	A SQuaRE-based Software Quality Evaluation Framework and its Case Study	Proposed a (Systems and software Quality Requirements and Evaluation) SQuaRE-based software quality evaluation framework, which successfully concretized many	They did not apply the framework to various domains to confirm its general applicability.

		product metrics and quality-in-use metrics	
(Nicchiotti *et al.,* 2016)	Machine Learning Strategy for Fault Classification using only Nominal Data	Implemented Gaussian Mixture Model (GMM), Support Vector Machines (SVM) and Auto Associative Neural Networks (AANN) into a framework meant for fault classification and diagnostics.	Fault classification is possible when only "healthy" data is used for training; healthiness of data remained vague.
(Nwandu and Asagba, 2017)	Automated Software Testing for Reliable System Development	Presented an automated model that tests the reliability of software systems and uses different metrics to estimate the reliability value.	The model employed only a few metrics and also did not accommodate other software quality attributes such as functionality, portability and efficiency.

(Hammouri *et al.,* 2018)	Software Bug Prediction using Machine Learning Approach	Created an evaluation model after assessing the efficiency of three machine learning algorithms (– NB, DT and ANNs) in fault prediction	Historical data needed for the mode's functionality may not be easily available.
(Pattnaik *et al.,* 2018)	Prediction of Software Quality Using Neuro-Fuzzy Model	Proposed a hybrid model (– using fuzzy logic and neural network) that predicts software quality based on the effects of certain specific features observed during software development.	Prediction was based on some specific features that are harnessed during the development process.
(Pattnaik *et al.,* 2019)	Software Quality Prediction Using Fuzzy Logic Technique	Applied fuzzy logic to create a model that largely recognizes quality factors in software quality.	Only quality factors were simulated and quantified.

(Bajpai *et al.*, 2019)	Conjecture of Scholars Academic Performance using Machine Learning Techniques	Applied machine learning in investigating students' details with the intent of assessing their academic performance	Analysis and assessment focused on students' performance other than the system that performs the assessment.
(Li *et al.*, 2019)	Evaluating Software Defect Prediction Performance: an updated benchmarking study	Proposed a metric-based evaluation model that predicts the presence of faults in software code	The model is predictive and is driven by a white-box evaluation which may be difficult in terms of code availability.
(Kim *et al.*, 2020)	Unexpected Collision Avoidance Driving Strategy Using Deep Reinforcement Learning	Applied reinforcement learning in analysing and predicting injury severity in unexpected traffic signal violation at an intersection	Reinforcement learning was applied to traffic control.
(Tsantekidis *et al.*, 2020)	Price Trailing for Financial Trading using Deep Reinforcement	Applied reinforcement learning in designing a data pre-processing	Model makes prediction on a traded asset.

	Learning	framework that trades forex currencies.	
(Kopyltsov, 2020)	Selection of Metrics in Software Quality Evaluation	Proposed an expert system that seeks to find improvement measures in software quality using quality metrics.	Quality improvement was prioritized with less attention to the measurement of the quality that ought to be improved.
Moreno, *et al.*, (2020)	Application of machine learning techniques to the flexible assessment and improvement of requirements quality	Applied machine learning in classifying set of requirements according to their quality and automated the process of inferring and applying the adapted quality rules.	The work focused on the quality of requirements instead of that of the system itself and stated no criteria for the assessment of the requirements quality.
(Kassie and Singh, 2020)	A Study on Software Quality Factors and Metrics to Enhance Software Quality Assurance	Surveyed for quality factors whose ranking of ordering suggests it's level of criticality for improving software quality.	Software quality factors were ranked on the basis of user responses and survey feedback.

(Cowlessur *et al.,* 2020)	A Review of Machine Learning Techniques for Software Quality Prediction	Studied various machine learning techniques to ascertain their performances in software quality prediction, and expected that results from the techniques should be more efficient.	The survey observed that more techniques need to be used to explore in-dept into software quality prediction.
(Omri and Sinz, 2021)	Machine Learning Techniques for Software Quality Assurance: A Survey	Studied the role of deep learning algorithms in apportioning preference to test cases and in the prediction of faults early enough during software development.	The survey focused on adherence to program semantics and structure, strictly for the purpose of system maintenance.
(Chen *et al.,* 2022)	Evaluating Code Improvements in Software Quality Course Projects	Presented a metric-based analytic evaluation of students' coding projects.	Investigated the effects of software quality course on the quality of projects.

2.5 Direction of the Research

Whereas extant literature suggests that previous works are not exhaustive enough, (Omri and Sinz (2021) focused on system maintenance rather than system quality, Cowlessur *et al.* (2020) discovered insufficient exploration into intelligent software quality prediction, Moreno *et al.* (2020) focused on the quality of requirements, Pattnaik *et al.* (2019) only simulated quality factors, Li *et al.* (2019) focused on predictive detection of fault rather than its measurement based on system performance, Nwandu and Asagba (2017) focused on estimating the reliability value of given software, Rana *et al.* (2015) focused on quality categorisation, and Chen *et al.* (2011) is limited to only four products), this work extends fully to the evaluation of given software with different software metrics, employing machine learning technique namely reinforcement learning technique for an intelligent and reliable evaluation. The evaluation takes cognisance of other software quality attributes based on six main quality attributes as identified by the ISO 9126-1 standard - Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability. The different software metrics employed in the work are subject to these six quality attributes. Reinforcement learning is chosen since available models did not utilize it.

The reinforcement learning method aims at using observations gathered from the communication with the environment to take actions that would maximize the reward or minimize the risk. Reinforcement learning is a learning method in

which decisions are made on the basis of what next actions are to be taken such that the outcome is more positive (Ayon, 2016). The agent does not have a prior idea of which actions to take until it has been given a situation. The method and operation of the action taken by the agent usually have effect situations and their future actions. The operational functioning of reinforcement learning is dependent on two criteria: trial-and-error search and delayed outcome. Reinforcement learning is a type of machine learning which learns a strategy to tackle a task based on an observation gathered from the surrounding world (Ayodele, 2010). In order to produce intelligent programs (also called *agents*), reinforcement learning goes through the following steps (Mohssen *et al.*, 2017):

1. Input state is observed by the agent.
2. Decision making function is used to make the agent perform an action.
3. After the action is performed, the agent receives reward or reinforcement from the environment to update its knowledge.
4. The state-action pair information about the reward is stored.

The information kept in memory serves as a tool for fine-tuning the next policy for a particular state in terms of action (i.e. evaluation of its last action). This helps the agent to make optimal decision on the best policy for a task. This indicates that every action taken by the agent has environmental impact and the learning agent is guided by the feedback that comes from the environment (Ayodele, 2010).

CHAPTER THREE

METHODOLOGY

3.1 Methodology of the Study

The research area of software evaluation can be viewed as a continuous trend within the software engineering circle. This is because as more innovations are made in technology, leading to the development of more complex and sophisticated software systems to move along with current trends, there is a continuous need for software evaluation in order to provide best solutions to user needs. This work presents an intelligent method for assessing the different software attributes with the aim of performing actual evaluation of the quality of the given software considering the number of functional units in the system and the duration of testing. The study adopted Extreme Programming (XP), an Agile framework which encourages simplicity (develop what is required and nothing more, with minimal lines of code), code and test often to make necessary changes when needed which helps to realize fault-free software. The design of the model is presented in object-oriented platform (for MySQL Server Databases). The model is systematically developed using

i. Object oriented analysis and methodology

ii. Use case analysis

iii. Simulation.

Object oriented analysis is usually utilized in modeling real-world system requirements, independent of the implementation environment. The real-world modeling using object-oriented analysis is done in more complete fashion than traditional methods. This is because objects are organized into classes and the model is based on objects rather than data and processing.

Object oriented methodology on the other hand is a development approach used in building software systems with an intention of ensuring reusability. Its reusability characteristic is unique and essential because new/created object automatically inherits the data attributes and characteristics of the class from which it was originated and also inherits data and behaviour from all super-classes in which it participates. The methodology inspires software developers to think of a problem in terms of the application domain early enough and apply a consistent approach throughout the entire life-cycle. This is because the methodology allows for improved reliability and flexibility such that new object behaviours can be built from existing objects due to the fact that objects can be dynamically called and accessed. In addition, object-oriented methodology has an advantage of reduced software maintenance because most of the processes in the system are encapsulated, hence the behaviours may be reused and incorporated into new behaviours.

Use cases are applied to capture the intended behaviour of the system to facilitate the identification of operations that support the testing process.

3.2 Analysis

Analysis is generic to every field of study especially where development processes are followed. Analysis can be described as the process of studying a model or procedure with the aim of identifying its intended goals and purposes and further create systems and procedures that will achieve the goals in a systematic and efficient manner. Ideally, analysis can be viewed as a problem-solving technique which subjects a system into some form of breakdown into its component units. Typically, this breakdown process is done for the purpose of studying how well those components interact and cooperate among themselves in order to accomplish their common goal. In other words, analysis may be defined as a procedure by which an intellectual whole is broken down into its component parts. Therefore, analysis can be perceived as a decision maker. This is because the results of a preceding analysis usually serve as a building-block for a new system which may now be extended to include more functionality.

3.2.1 Analysis of the Study Models

This study conducted an in-depth analysis of four (4) existing models due to their individual closeness to the idea conceived in the new model. The analysed works belong to Bindal (2013), Chen *et al.* (2011), Rana *et al.* (2015) and Nwandu and Asagba (2017). The analysis was targeted at the models' schematic

diagrams, sequence of execution (flowchart), and implementation algorithms. These helped to expose the limitations of the works through which the motivated study was fine-tuned in line with the gap identified at problem identification. The analysis of the existing models is given thus:

Bindal (2013) carried out an investigatory study on how Markov Mode of software is drawn, its application and how it can be used in estimating the reliability of software. The software testing model as proposed by Bindal (2013) is diagrammatically illustrated in the framework shown in Figure 3.1:

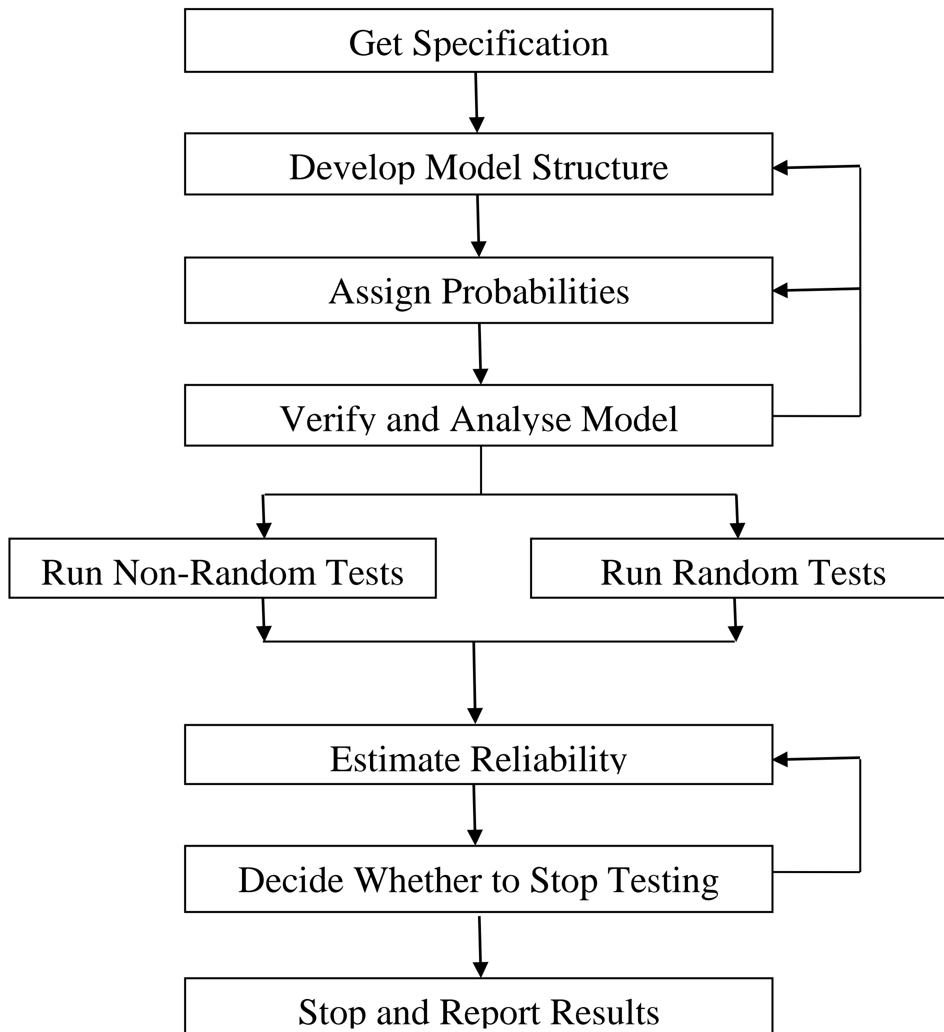


Figure 3.1: Framework of Bindal (2013) Model.

Figure 3.2 illustrates the step-by-step operations of Bindal (2013) testing system using flowchart.

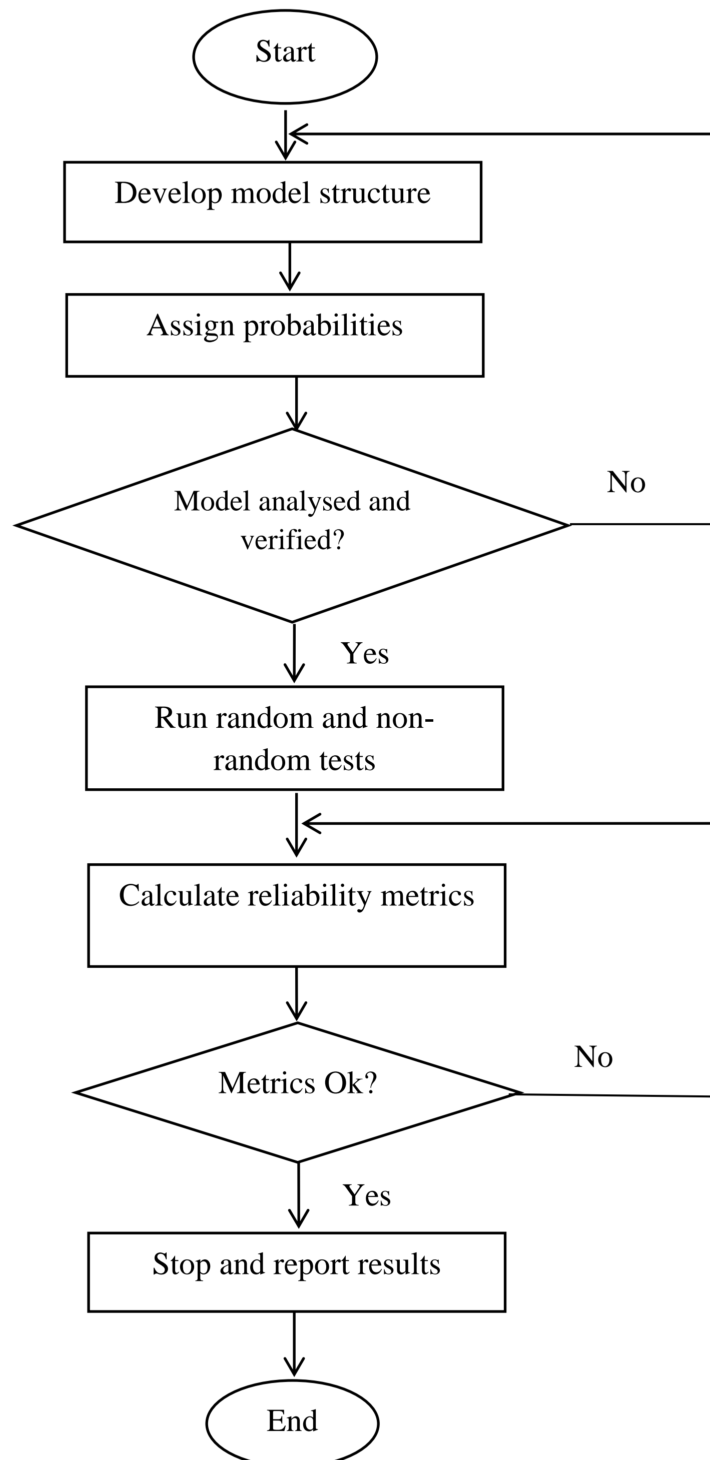


Figure 3.2: Flowchart of Bindal (2013) Model.

3.2.1.3 Algorithm of Bindal (2013) Model

Step 1 Initialisation (component, test suite, probability, MTTF, MTTR, MTBF, Availability, result)

Step 2 For each component

Generate Test suite

Step 3 For each test suite

Assign probability

Run random test

Run non-random test

Step 4 Calculations

Mean Time To Failure (MTTF)

Mean Time To Repair (MTTR)

Mean Time Before Failure (MTBF)

Availability

Step 5 Report result

Step 6 End

3.2.1.4 Limitations of Bindal (2013) Model

i. The model is only a measurement model to track failures at different time instants.

ii. Specification was based on input parameters. This may not give the required evaluation.

iii. The model only gave a description of how Markov model can be applied in failure detection using metrics but failed to implement the actual reliability assessment.

Chen *et al.* (2011) presented a semi-automated approach to software process evaluation using machine learning techniques. Figure 3.3 illustrates the framework of Chen *et al.* (2011) semi-automated approach to software process evaluation.

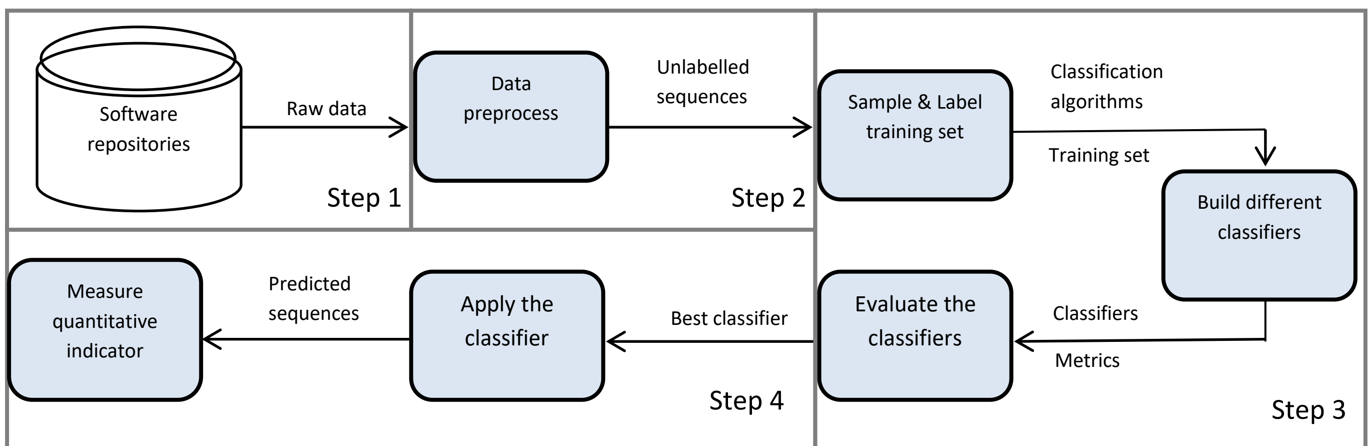


Figure 3.3: Framework of Chen *et al.* (2011) Model.

Figure 3.4 shows the systematic flow of processes in Chen *et al.* (2011) proposed model.

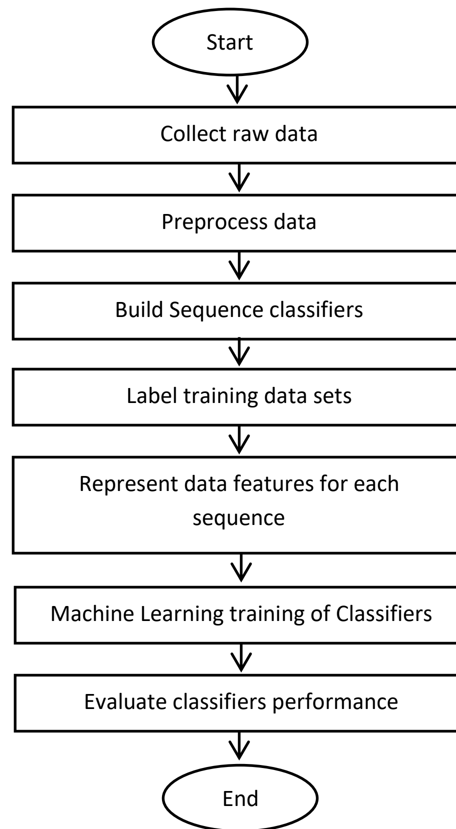


Figure 3.4: Flowchart of Chen *et al.* (2011) Model.

3.2.1.5 Algorithm of Chen *et al.* (2011) Model

Step1: Initialisation (Data, DataSet, S_1 … S_N, True Positive (TP), False Positive (FP), False Negative (FN), Precision, Recall, Process Execution Qualification Rate (*P*))

Step2: Data = *data*

For each Data do {

Data = S{1 … N}

Generate Unlabelled Sequence {S_1 … S_N}

{S_1 … S_N} = {Normal, Abnormal}

Step3: Call Classification Algorithm

Apply Classifier Metrics

Step 4: Evaluate Classifiers

Precision = TP / (TP + FP)

Recall = TP / (TP + FN)

F-Measure = (2*Precision*Recall) / (Precision + Recall)

Step4: Output P = (# predicated normal sequences * Precision) / (total # sequences * Recall)

Step5: End

3.2.1.6 Limitations of Chen *et al.* (2011) Model

i. Model is limited to four software products only.

ii. It is uncertain to extend and apply the framework to more complicated processes.

iii. Model made some "precision" assumptions which may not hold in real-world situations.

Rana *et al.* (2015) applied Machine Learning approaches namely Pattern Recognition and Classification to predict a quality category in which a given software falls, during its development. The sequence of activities involved in Rana *et al.* (2015) model is contained in the framework shown in Figure 3.5.

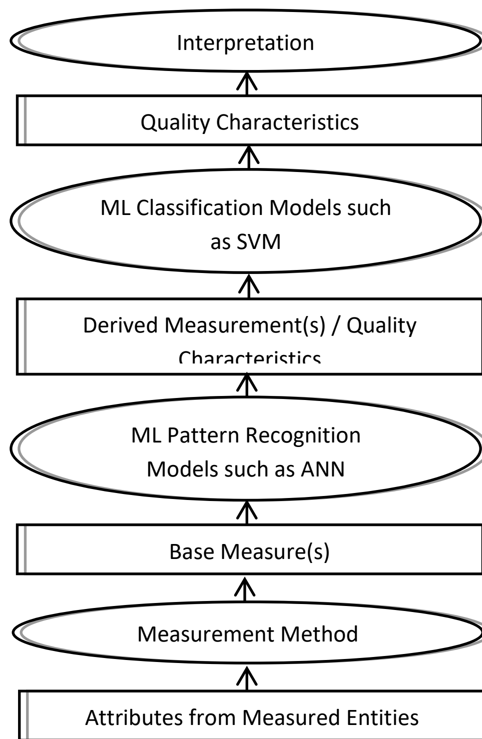


Figure 3.5: Framework of Rana *et al.* (2015) Model.

Figure 3.6 shows the systematic flow of processes in Rana *et al.* (2015) proposed quality categorisation system.

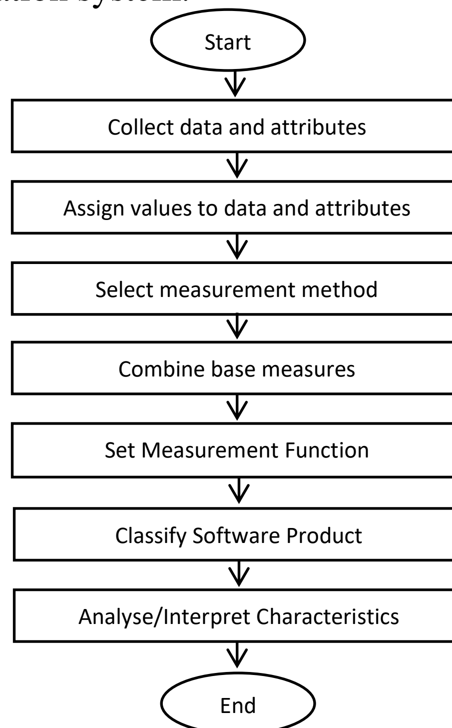


Figure 3.6: Flowchart of Rana *et al.* (2015) Model.

3.2.1.7 Algorithm of Rana *et al.* (2015) Model

Step1: Data Collection

Get attributes $a_1 \ldots a_n$

Identify Measurement Method

Select base measure

Step2: Data Preparation

Define Measurement Function:

Combine base measures

Set derived measure

Step3: Data Analysis

Select Analysis Model:

Combine base and derived measures

Interpret obtained quality

Step 4: End

3.2.1.8 Limitations of Rana *et al.* (2015) Model

i. Model focused on quality Categorisation

ii. Model utilized a few metrics

Nwandu and Asagba (2017) designed an automated framework on how software can be investigated via testing and how it is used for estimating the reliability of

software. The software testing process as proposed by Nwandu and Asagba (2017) is illustrated in the framework shown in Figure 3.7:

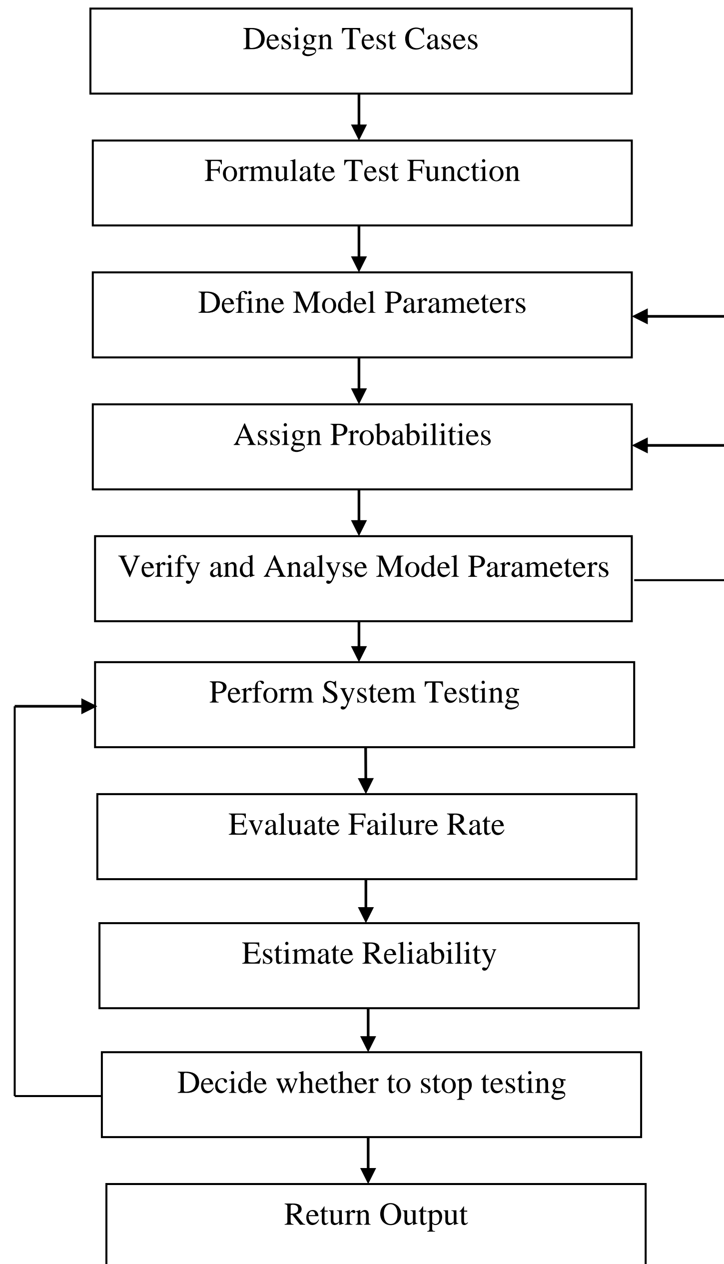


Figure 3.7: Framework of Nwandu and Asagba (2017) Model.

Figure 3.8 illustrates the step-by-step operations of the system proposed by Nwandu and Asagba (2017) using flowchart.

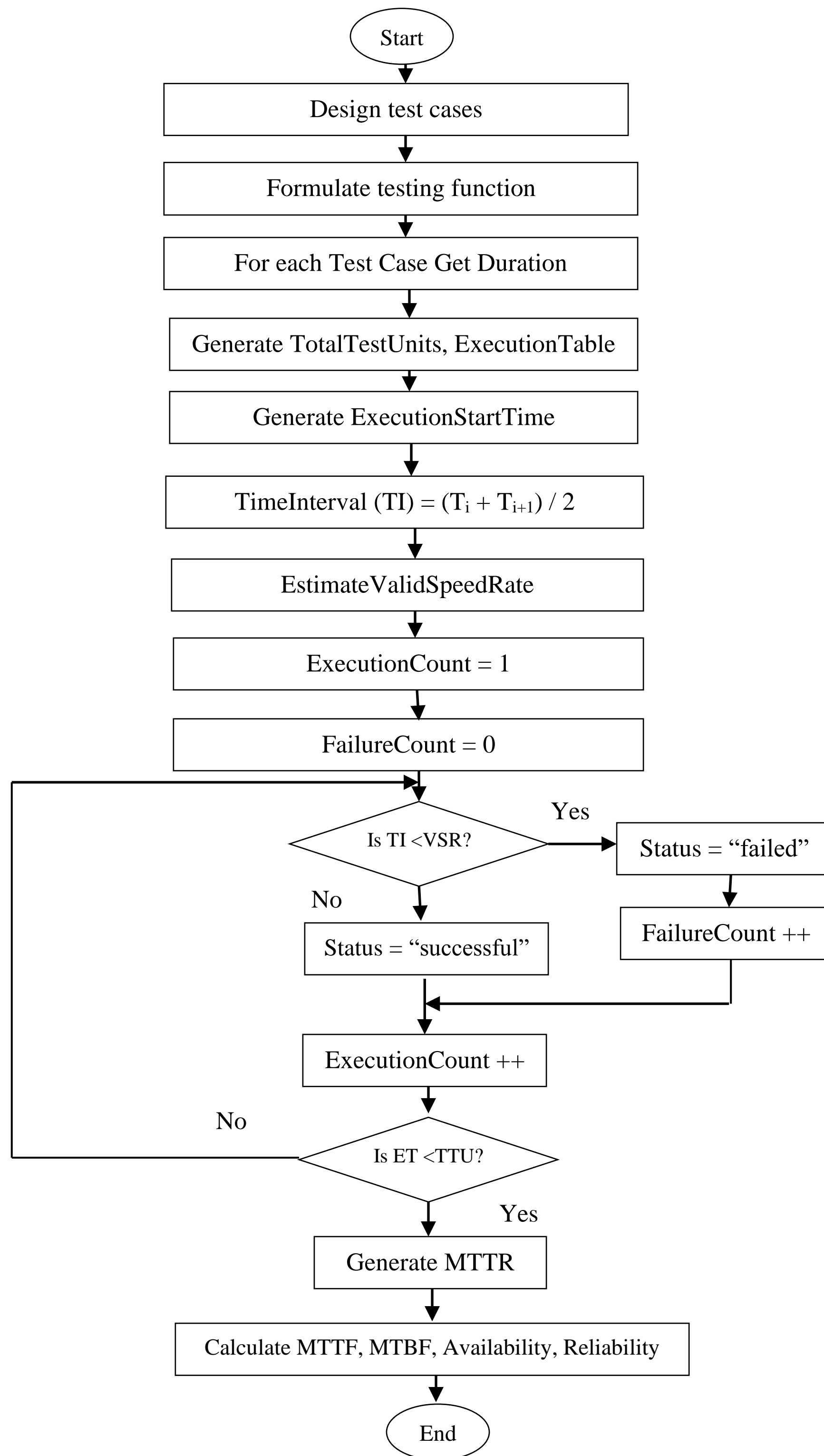


Figure 3.8: Flowchart of Nwandu and Asagba (2017) Model.

3.2.1.9 Algorithm of Nwandu and Asagba (2017) Model

Step 1 Initialization (TestCases, TotalTestUnits, Duration, ExecutionStartTime, ExecutionEndTime, ValidSpeedRate, ExecutionCount, TimeInterval, FailureCount, Status, MTTF, MTTR, MTBF, Availability, Reliability, Create execution tables)

Step 2 Do {

Formulate Test Function

Status = “successful” || “failed”

Step 3 For each TestCase (1 to n)

Get Duration

Generate TotalTestUnits (TTU)

GenerateExecutionStartTime (T_i)

GenerateExecutionEndTime (T_{i+1})

Generate ExecutionTable (ET)

Step 4

Calculate TimeInterval (TI) = (T_i + T_{i+1}) / 2;

Estimate ValidSpeedRate (VSR)

ExecutionCount = 1;

Step 5 If TI < VSR

Status = “failed”;

Increment FailureCount

Increment ExecutionCount

Step 6 Else

Status = “successful”;

Increment ExecutionCount

Step 7 If ET < TTU

Generate Mean Time To Repair (MTTR)

Calculate Mean Time To Failure (MTTF), Mean Time Between Failure (MTBF), Availability, Reliability

Step 8 End

3.2.1.10 Limitations of Nwandu and Asagba (2017) Model

i. The specification used in the model was based on input parameters. This may not give the required evaluation.

ii. The model only gave a description of how reliability of software can be assessed using metrics but failed to implement the assessment of other software quality attributes.

iii. The model utilized only a few numbers of metrics in its course of performing/actualizing its major objective of software reliability estimation.

3.3 Design

Design is a process in which an agent formulates a specification of a software artefact with the intent of accomplishing certain goals. A software requirement specification document reveals what the system does and becomes input to the design process which indicates how the software system works. Design process involves problem solving and planning of a software solution. The process includes both low-level component and algorithm design and a high-level

architecture design. A high-level design is prepared after answering questions of requirements. Design is then validated against requirements.

Design can be described as a sequence of steps that enables the software engineer to describe all aspects of the intended software. It starts with initial requirement and ends with the functional design. This makes design a highly significant phase in software development because the designer plans how the software should be produced in order to make it functional, reliable, and reasonably easy to understand, modify and maintain. Design is usually modified on regular basis until it is finally documented to produce software design document.

3.3.1 System Modeling

The system evaluates given software in an intelligent manner to ascertain the quality of the software product. On determining the number of functional units, the system applies reinforcement learning to generate test policy from formulated test function upon which the system testing process depends. The system evaluates the software in terms of speed, accuracy and precision such that, the system returns “execution successful” if process is executed within a minimal (optimal) time interval or otherwise returns “system error”. A built-in counter for one result updates itself at the end of each execution depending on the outcome whereas the general execution counter records an increment at every single execution. The various built-in metrics corresponding to different

software quality attributes utilize the results to evaluate the software. Software artefacts used in the design of the system are:

i. Flowchart
ii. Use Case Diagram
iii. Data Flow Diagram
iv. Sequence Diagram
v. Entity Relationship Diagram
vi. Class Diagram

3.3.2 System Model Architecture

The designed software evaluation system is illustrated in the simple architecture shown in Figure 3.9.

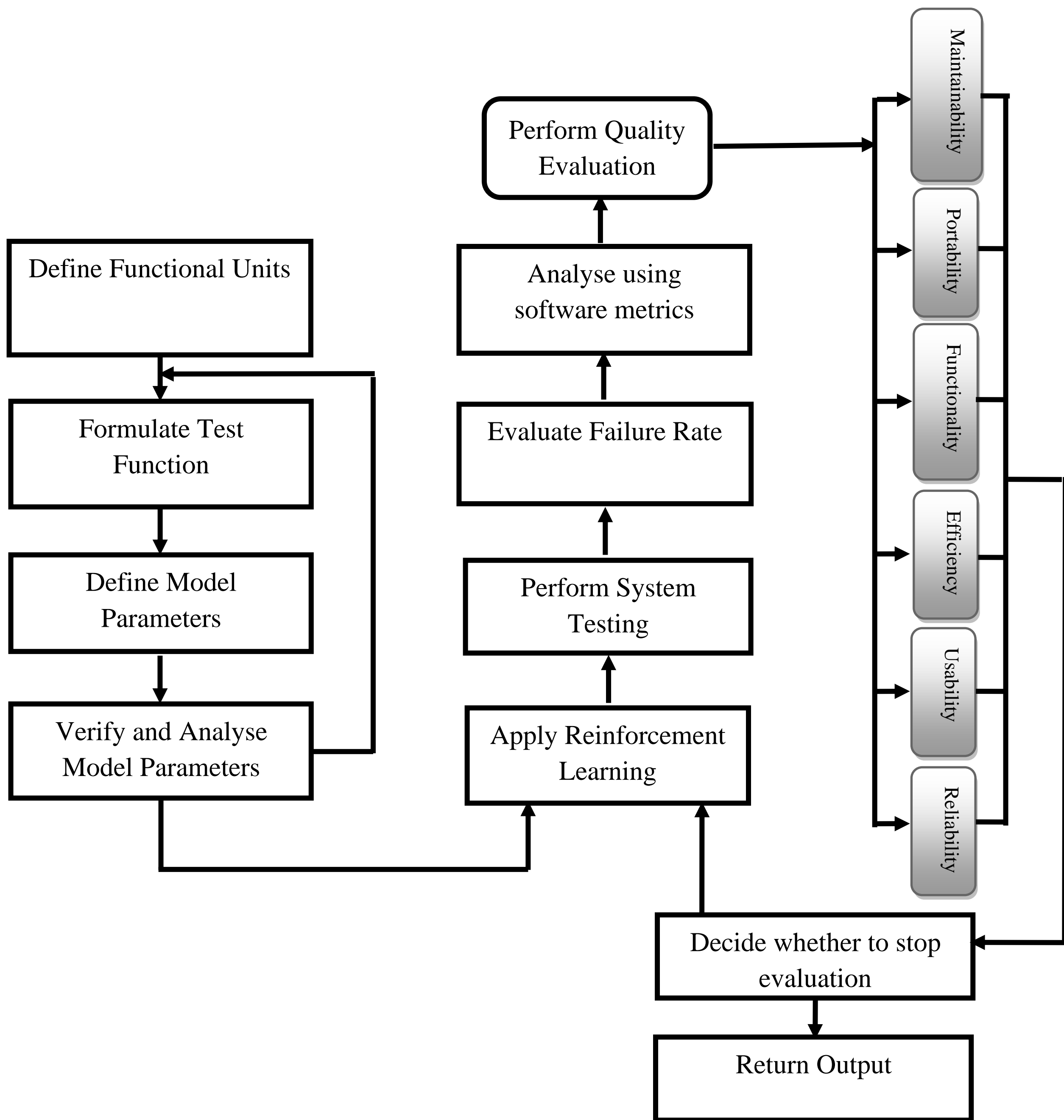


Figure 3.9: Architecture of the Model.

3.3.3 Advantages of the System

i. Simple to use

ii. Reduced evaluation time via intelligent decision

iii. Increased reliability and confidence due to clarity of results

iv. Concise but exact coverage of code resulting to effective detection of defects
v. Reusability of test cases

3.3.4 Expectations of the System

i. Easy data collection approach
ii. Effective statistical assessment of software quality attributes
iii. Applicability to various software environments
iv. Reliable rating of software performance of intended functions

3.3.5 Assumptions of the System Model

The model assumes that the execution time of software being evaluated is indicated by some randomly generated time instants, noting the intervals between the time instants as the software execution speed.

The model assumes that the longer the interval, the slower the software executes and the higher the tendency of having a system failure. In other words, the shorter the interval, the faster the rate of the software execution, indicating higher accuracy in evaluated software's intended functionality.

3.3.6 Performance Measures of the System Model

The model applies reinforcement learning to generate a set of suitable test policies based on the indicated user-defined performance objectives (or test functions) of the software being evaluated perceived through the number of

functional units. Higher speeds of execution returns "execution successful" status whereas very low speeds return "system error" status due to interruptions from subsequent emergence of time instants that are auto generated by the model.

Each execution time is recorded in a database. The model provides step-counters for both successful and error executions. These records are used in the assessment of the different software attributes using their associated metrics.

Software quality is evaluated using the metric values of the attributes. The numeric outcomes of the metric assessment are thereby ploughed into a formulated mathematical model to evaluate the quality of the software. The formulated mathematical model meant for this purpose is given thus:

$$Quality_{eval} = \left(\sum_{i=1}^{n} \frac{a_i}{n-1}\right) * \left(\frac{\ln(n)}{(n-1)^{1.25}}\right) \tag{3.1}$$

where a_i = {reliability, usability, efficiency, functionality, maintainability, portability}

and therefore, $n = 6$.

The denominator of the n's controlling unit of the model equation contains an exponent of 1.25 called unifying factor. The unifying factor is used to harmonize $Quality_{eval}$ to unity, such that a value of 1 indicates perfect quality.

Reliability Evaluation:

The interval between one failure time and the other is continually captured and these intervals are used to determine the Mean Time to Failure (MTTF) of the software being evaluated. MTTF is given by:

$$\text{MTTF} = \sum_{i=1}^{ne} \frac{t_{i+1} - t_i}{ne - 1} \quad (2.3)$$

where n_e is the number of failures and

t_i is the time instants at which failure occurs.

Other built-in reliability metrics are:

i. Mean Time to Repair (MTTR) = $\frac{ne}{n}$, (2.4)

where n is the total number of software executions.

ii. Mean Time Between Failures (MTBF) = MTTF + MTTR (2.5)

iii. Availability = $\frac{\text{MTTF}}{MTBF} * 100$ (2.6)

The system then estimates the Reliability (R) of the software that is being evaluated using

$$R = 1 - \frac{ne}{n} \quad (2.7)$$

where n is the total number of software executions.

Usability Evaluation:

The system measures the usability of the software in review by analysing the number of executions completed (i.e. ExecutionCount) and the numbers of successes and errors (SuccessCount and ErrorCount) recorded by the software. It therefore records usability as the ratio of SuccessCount to ExecutionCount.

Efficiency Evaluation:

The system measures the efficiency of the software using the Root Mean Square Error (RMSE) which is given by the square root of the mean square error which measures the average magnitude of error:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{\text{E}} - \text{E})^2} \quad (2.12)$$

$n =$ number of executions, $\bar{\text{E}} =$ actual functional units and

$\text{E} =$ executed functional units

Another efficiency metrics built into the system is Mean Magnitude Relative Error (MMRE) otherwise known as Mean Absolute Relative Error (MARE) is calculated to assess the rate at which the software undergoing evaluation performs its intended functions. MMRE is given by:

$$\text{MMRE}\,(\%) = \frac{1}{n}\sum_{i=1}^{n}\text{MRE} * 100 \quad (2.8)$$

where $\text{MRE} = \frac{|\bar{\text{E}} - \text{E}|}{\bar{\text{E}}}$

Functionality Evaluation:

The system quantifies what the software under review performs by counting the number and types of features/functions used in the software application. This is known as Function Point (FP) and is given by:

$$\text{FP} = \text{Count-total} * \text{CAF} \qquad (2.13)$$

where $\text{CAF} = \left[0.65 + 0.01 * \sum (f_i)\right]$

$i = 1 \text{ to } 14$

Maintainability Evaluation:

The system measures the maintainability of software by generating the amount of effort (time) required to fix all faults found in the software. This time effort is known as Technical Debt (TD) which is equal to the Mean Time To Repair (MTTR) used in reliability evaluation. Also, Technical Debt Density (TDD) is also employed by the system as a ratio of TD to the source lines of code in the software to measure maintainability. TDD is given by:

$$\text{TDD} = \frac{\text{TD}}{\text{LOC}} \qquad (2.14)$$

Portability Evaluation:

Portability measurement is strictly a platform issue. The system evaluates the portability of software by its ability to function on various system platforms at minimum varying runtimes. It measures the software's average installation time and its ability to be removed and reinstalled if need be.

$$\text{Portability} = \frac{Number\ of\ successful\ ports\ * 100}{Total\ number\ of\ ports} \tag{2.17}$$

where

Total number of ports include browser and version, operating system and version, programming language, processor make and speed, software modules (units) etc. available within the operation environment.

Successful ports are the ports used by the software within the operation environment.

Application of Reinforcement Learning in this model:

This study demonstrated the evaluation model as it applies reinforcement learning using the general process for reinforcement learning shown in figure 3.10.

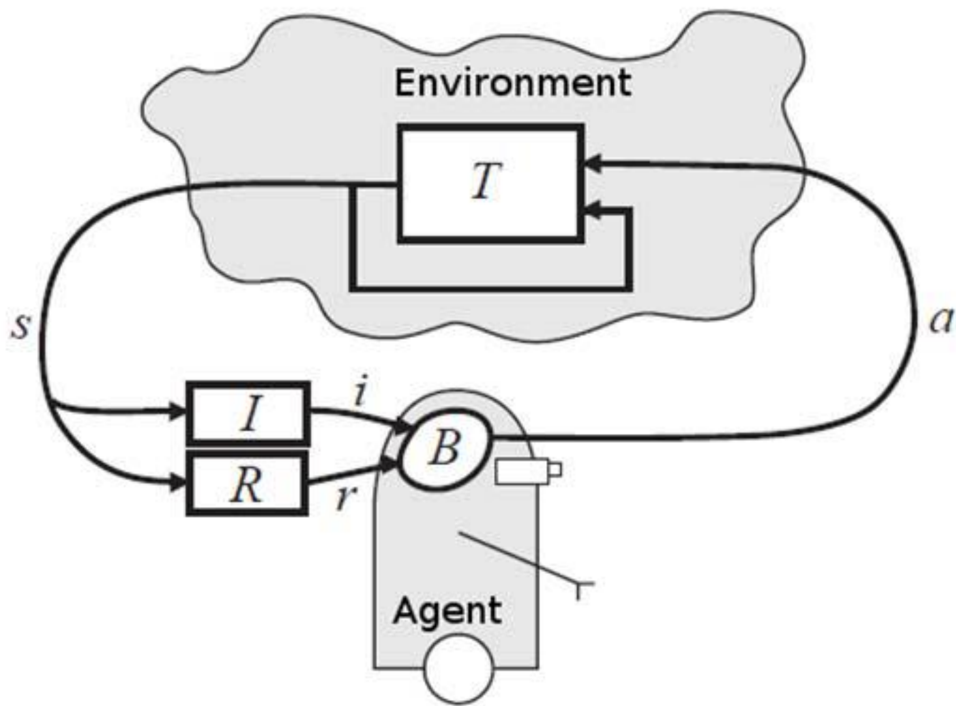


Figure 3.10: The Reinforcement Learning Model (Kaelbling *et al.*, 1996).

In Figure 3.10, the agent *(our model)* receives an input *i (number of functional units)*, current state *s (first to final sub-functional units)*, state transition *r (initial, intermittent, or final testing)* and input function *I (objective function)* from the environment. Based on these inputs, the agent *(our model)* generates a behaviour *B (test policy)* and takes an action *a (testing process)* which generates an outcome *R (successful or failure).*

Justification and Adoption of Reinforcement Learning in the Model:

Reinforcement learning is a unique supervised learning because the sample data-set does not train the agent. Instead, it adopts trial and error technique in updating its knowledge. The agent strengthens the process by making several

decisions which rightfully takes care of the problem. In reinforcement learning, the model does not require any form of advice on the approach to be followed in solving the task, i.e., creation of execution policy. The model independently determines how an execution is performed for proper optimization of the reward, commencing execution with random testing and advanced methodologies.

In the real sense, reinforcement learning does not require much historical data nor labelled data as against supervised learning. It typically determines and makes use of value function based on the self-determined execution policy decided for an action. In our context, we adopt the Markov Decision Process (MDP) model which is the basis of reinforcement learning:

i. An Environment E and agent states S

ii. A set of actions A taken by the agent

iii. P(s,s') => $P(s_{t+1}=s'|s_t=s, a_t=a)$ is the transition probability from one state s to s'

iv. R(s,s') – Immediate reward for any action

In translation, our model takes the following MDP format:

i. Agent – An agent A that works in Environment E

ii. Action – Begin/Terminate/MeanTime/TimeCompare

iii. States – Sub-functional Unit values

iv. Reward – Successful/Failure

The proposed model in this study adopts on-policy and model-free mechanisms such that, the agent learns the value based on its current action *a* derived from the current strategy it employs to determine next action as well as requires not to store all the combination of states and actions since it relies on trial-and-error to update its knowledge. The series of actions required to be taken by the agent include:

i. The Agent randomly generates a valid execution speed (VES) within a stipulated value scope (as its test standard for evaluating every software).

ii. Execution time of software is indicated by some randomly generated time instants (based on the number of functional units), noting the intervals between the time instants as the software execution speed and compares them (the execution speed) with the VES.

iii. The shorter the time interval, the faster the rate of the software execution, indicating accuracy in evaluated software’s intended functionality. The model returns “Successful Execution”.

iv. The longer the time interval, the slower the software executes and the higher the tendency of having a system failure. The model returns "System Error".

The agent's environment triggers the agent by sending a state (Number of functional units) to it. Based on its knowledge, the agent takes actions (i. to iv.) in response to that state. This is followed by the environment's messaging that sends a pair of next state and reward (system status) as a feedback to the agent. The returned reward serves as the agent's basis for further updating of its knowledge and evaluating its most recent action. This continues to iterate until a final state is communicated from the environment to the agent to bring a round of execution to an end.

3.3.7 Use Case Diagram of the System Model

The use case diagram shown in Figure 3.11 models the intelligent software evaluation system proposed in this study.

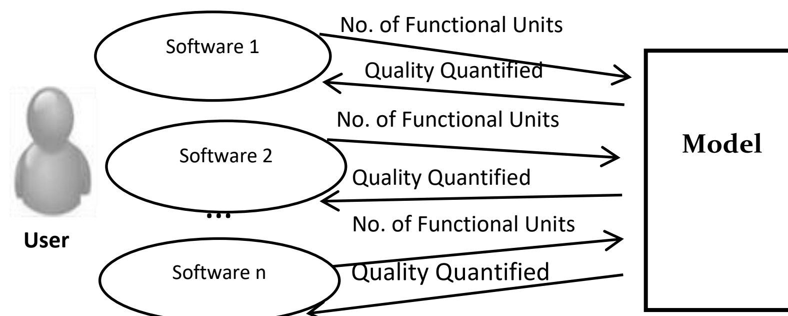


Figure 3.11: Use Case Diagram of the Model.

Figure 3.11 represents the proposed model in a simplified design in which a user inputs the number of functional units for each software to be evaluated, which triggers the model into internal operations that eventually return some values to the user to represent the quality attributes in numeric quantities.

3.3.8 Data Flow Diagram of the System Model

The flow of data in the proposed intelligent software evaluation system is modelled in the diagram shown in Figure 3.12.

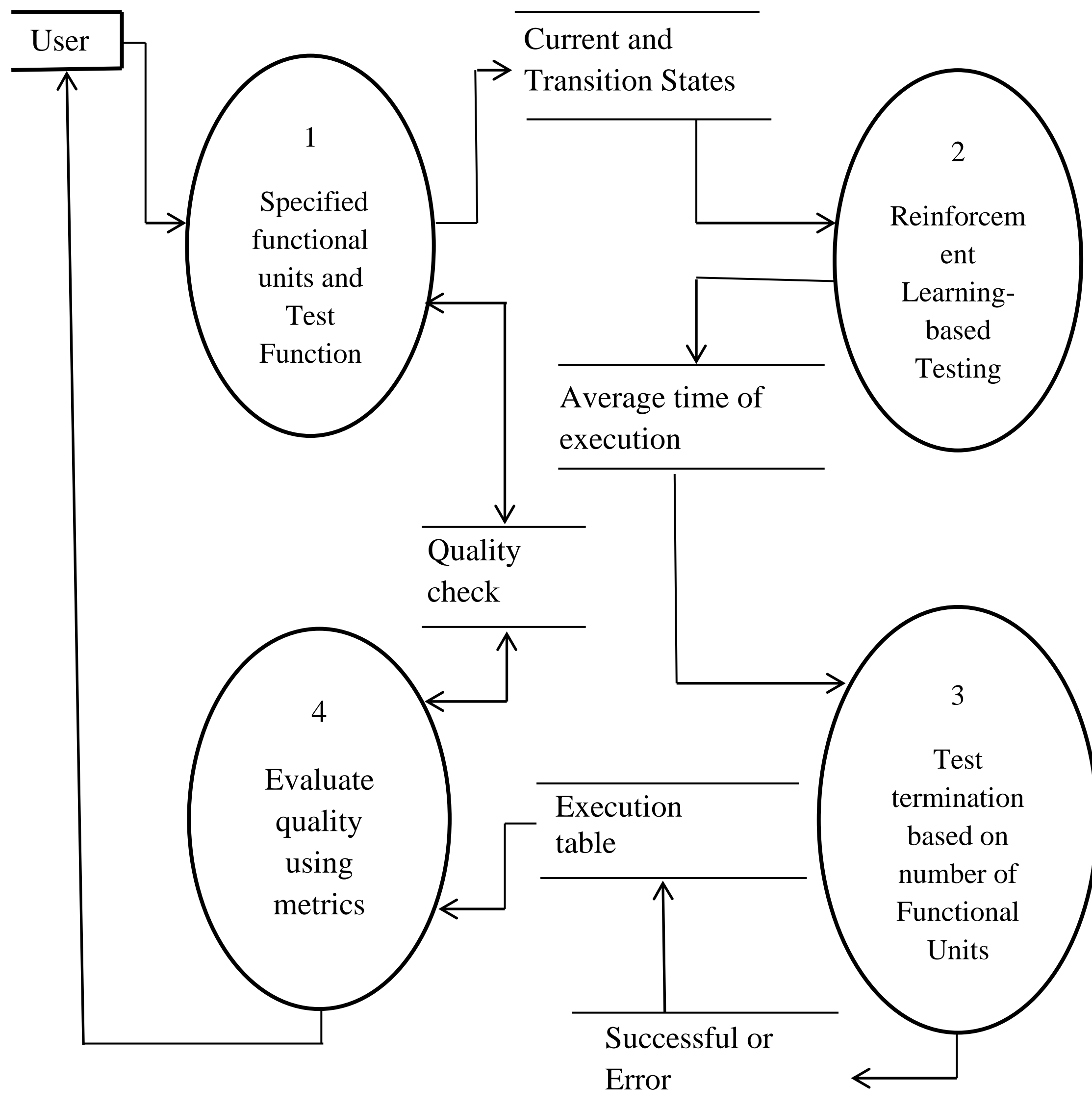


Figure 3.12: Data Flow Diagram of the Model.

Figure 3.12 presents the proposed model in a four-staged data flow arrangement. As the model receives the testing parameters (namely number of functional units), the specified test function allows for a reinforcement learning-based assessment having a knowledge of the system's current state. The outcomes of the assessment are stored in appropriate formats and are used to undertake a metric-based analysis of the system which gives the user an idea of the quality level of the software in review.

3.3.9 Sequence Diagram of the System Model

The systematic flow of executable activities among the participants in the proposed intelligent evaluation model is illustrated in the sequence diagram shown in Figure 3.13.

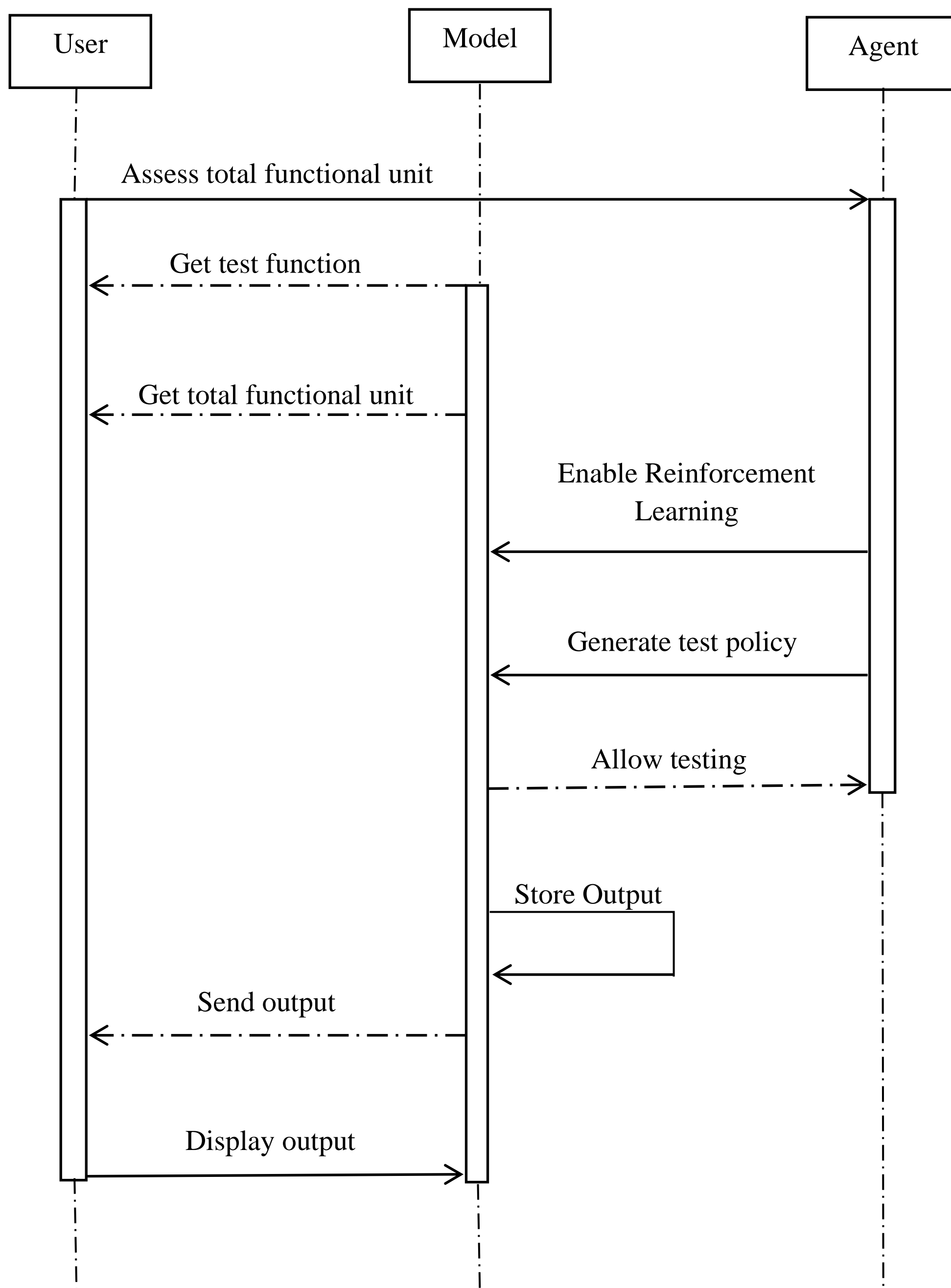


Figure 3.13: Sequence Diagram of the Model.

In Figure 3.13, the model is represented to contain three participants namely the user, the model itself and the embedded learning agent. As a middle man, the model transfers the number of functional units to the agent which enables the agent to perform a reinforcement learning assessment on the evaluated software.

The agent does this task using some independently determined policies that enables it to make the right decisions in testing the software functionality. The outcome of the exercise goes back to the user through the model's database.

3.3.10 Flowchart of the System Model

The flowchart of the proposed intelligent evaluation model is shown in Figure 3.14. The flowchart begins with the identication of the total number of functional units in the software to be evaluated and initialization of other parameters needed for the evaluation. The model transmits the parameters into a reinforcement learning agent which carries out a system test using artificial intelligence. The outcome of the test updates the appropriate counter, either ErrorCount or SuccessCount. However, the number of executions performed is tracked to correspond with the total number of functional units initially identified. At the completion of execution rounds, the results of the tests sre used to calculate values for some built-in metrics which are further used in quantifying the quality attributes as a means of measuring the software quality.

The following key explains the abbreviations contained in the flowchart:

MTTR – Mean Time to Repair.

MTTF – Mean Time to Failure.

MTBF – Mean Time before Failure.

MARE – Mean Absolute Relative Error.

RMSE – Root Mean Square Error.

FP – Focal Point.

TD – Technical Debt.

TDD – Technical Debt Density.

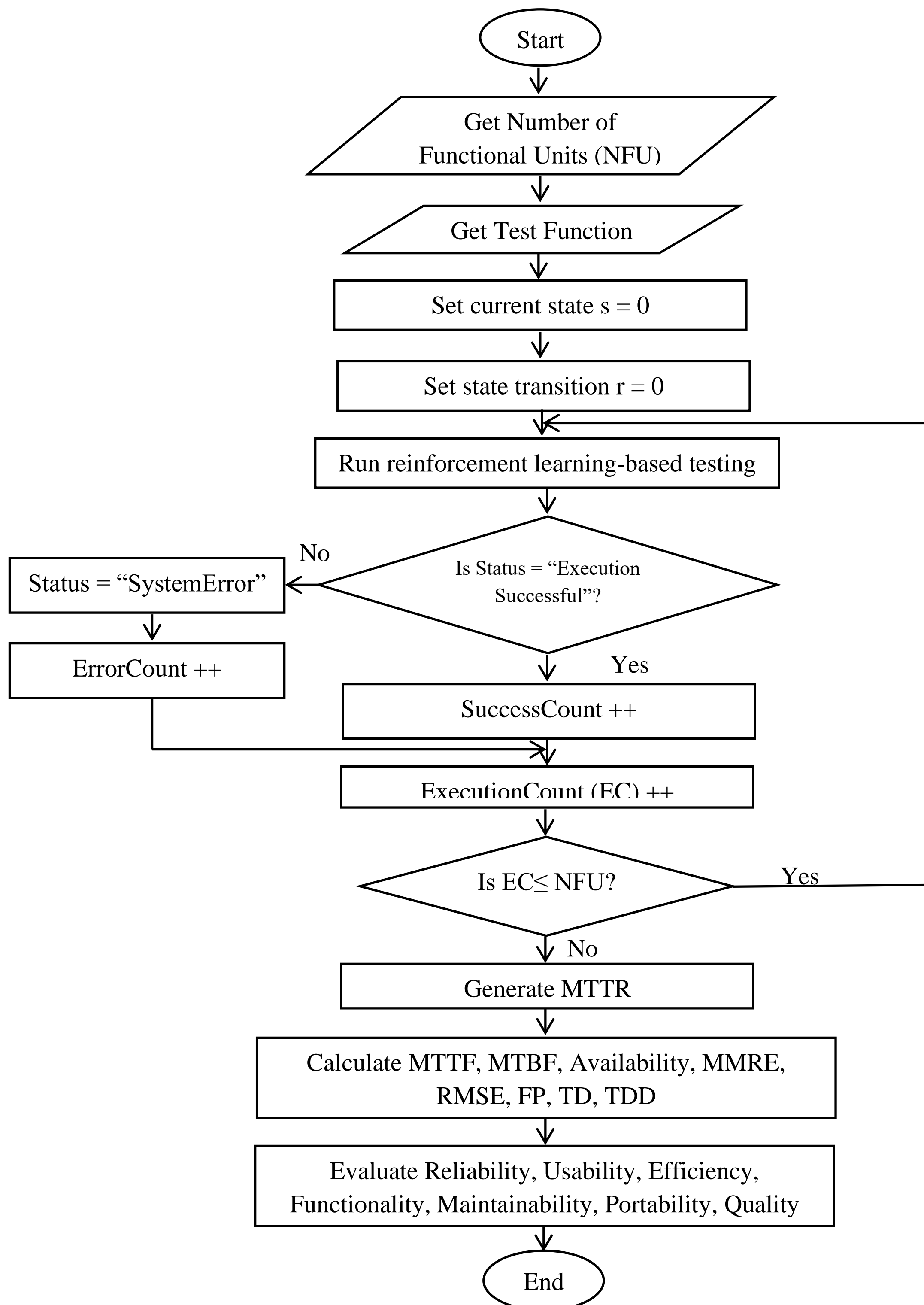


Figure 3.14: Flowchart of the Model.

3.3.11 Implementation Algorithm of the System Model

Step 1 Initialization (Test Function, TotalFunctional_Unit, Functional_Unit[1..n], states S [$s_0 \ldots s_n$] , actions A[$a_0 \ldots a_n$], transition state r, Execution_Start_Time, Execution_End_Time, Valid_Execution_Speed, Reward R, ExecutionCount, TimeInterval, SuccessCount, ErrorCount, Status, MTTF, MTTR, MTBF, Availability, MARE, RMSE, FunctionPoint, TechnicalDebt, TDD, Reliability, Usability, Efficiency, Functionality, Maintainability, Portability)

Step 2 Do {

Get TotalFunctional_Unit n

Specify Test Function

Status = "Execution Successful" || "System Error"

Step 3 **For** Functional_Unit 1 to n **do** {

s = 0

Reward Function R = S * A

Blackbox Transition Function T = S * A

Reward = 0

Call Reinforcement Learning algorithm module {

While s ≤ n do

a_1: Generate TestPolicy[1..n]

a_2: Generate Execution_Start_Time (T_i)

a_3: Generate Execution_End_Time (T_{i+1})

a_4: Calculate TimeInterval (TI) = (T_i + T_{i+1}) / 2;

a_5: Estimate Valid_Execution_Speed (VES)

ExecutionTime (ET) = TI

Reward = ET

Return Reward

}

ExecutionCount = 0;

SuccessCount = 0;

ErrorCount = 0;

Step 4 If Reward ≤ VES

Status = “Execution Successful”;

SuccessCount++

ExecutionCount++

Step 5 Else

Status = “System Error”;

ErrorCount++

ExecutionCount++

}

Step 6 If ExecutionCount ≥ TotalFunctional_Unit

Generate Mean Time To Repair (MTTR)

Calculate Mean Time To Failure (MTTF), Mean Time Between Failure (MTBF), Availability, Mean Absolute Error (MARE), Root Mean Square Error (RMSE), FunctionPoint, TechnicalDebt, Technical Debt Density (TDD)

Step 7 Else

GoTo **Step 3**

Step 8

Estimate Reliability, Usability, Efficiency, Functionality, Maintainability, Portability, Quality

Step 9 End do

Step 10 End

3.3.12 Input/Output Specifications

The specifications for the input and the output of the proposed intelligent evaluation system are shown in Tables 3.1 and 3.2 respectively.

Table 3.1: Input Specifications.

Field Name	FUNCTIONAL UNIT	TEST FUNCTION
Field Type	Integer	Text
Null or Not-null	Not-null	Not-null
Description	Integer	50 Characters

The model is designed to receive input values via a user-defined test function and the number of functional units. The model is configured to accept the number of functional units as integer values whereas the test function comes as text value formatted to contain no more than 50 characters as depicted in Table 3.1. On the other hand, the expected outputs which would be deposited in an

accessible repository are designed to be well-formatted according to the specifications illustrated in Table 3.2. The expected outputs include the time T_i at which the evaluation process begins, the time T_{i+1} at which the evaluation ends, the values calculated for each metric employed in the model, and the execution status of the model at any point in time, in addition to the number of functional units. The starting and ending times of the evaluation as well as the calculated metric values are stored as decimal values while execution status is returned as text with maximum character of 20.

Table 3.2: Output Specifications.

Field Name	FUNCTIONAL UNIT	START TIME (T_i)	END TIME (T_{i+1})	METRICS VALUE	EXECUTION STATUS
Field Type	Integer	Decimal	Decimal	Decimal	Text
Null or Not-null	Not-null	Not-null	Not-null	Not-null	Not-null
Description	Integer	Decimal	Decimal	Decimal	20 characters

3.3.13 Entity Relationship Diagram of the Model

The interactions between the system input and outputs are illustrated in the entity-relationship diagram shown in Figure 3.15.

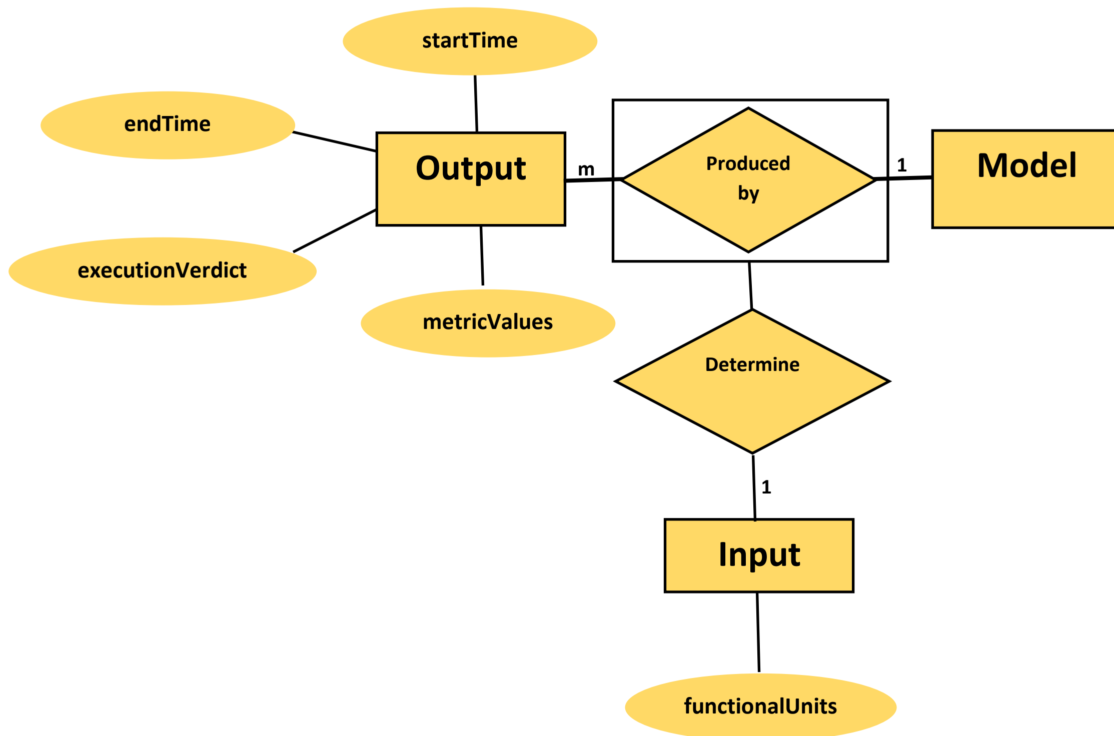


Figure 3.15: Model's E-R Diagram.

Figure 3.15 indicates that the number of functional units is a determinant to the various outputs that would be produced by the model as evaluation results. The outputs include the time instants at which execution begins and ends, the system status report and the numerical values for the metrics and the quality attributes.

3.3.14 Database Specifications

The database specifications showing the formats for data storage of the proposed intelligent evaluation system for both the user and the server are shown in Tables 3.3 and 3.4 respectively.

Table 3.3: User Table.

FIELD NAME	FIELD TYPE	NULL/NOT-NULL	DESCRIPTION
FunctionalUnit	Integer	Not-null	Long integer
StartTime[$T_{(i)}$]	Double	Not-null	Decimal (2dp)
EndTime[$T_{(i+1)}$]	Double	Not-null	Decimal (2dp)
Status	Text	Not-null	20 characters

The database contents are stored in the format as contained in Tables 3.3 and 3.4. Table 3.3 indicates that the output data alongside the number of functional units form the database data whose usage and manipulations are made by the user. The table further explains that the decimal values of the starting and ending times of evaluation are rounded to 2 decimal places whereas the number of functional units and the execution status report are stored in the format already described in Table 3.2. The Server Table 3.4 captures some important values calculated in the course of executing the evaluation including test duration formatted to 2 decimal places, total number of functional units, number of successful executions, number of failed executions, and total number of executions performed by the model formatted as long integer values.

Table 3.4: Server Table.

FIELD NAME	FIELD TYPE	NULL/NOT-NULL	DESCRIPTION
Test Duration	Double	Not-null	Decimal (2dp)
TotalFunctionalUnit	Integer	Not-null	Long integer
TotalSuccess	Integer	Not-null	Long integer
TotalFailure	Integer	Not-null	Long integer
TotalExecution	Integer	Not-null	Long integer

3.3.15 Class Diagram of the System Model

The model's database contents and their relationships are visually demonstrated in the class diagram shown in Figure 3.16.

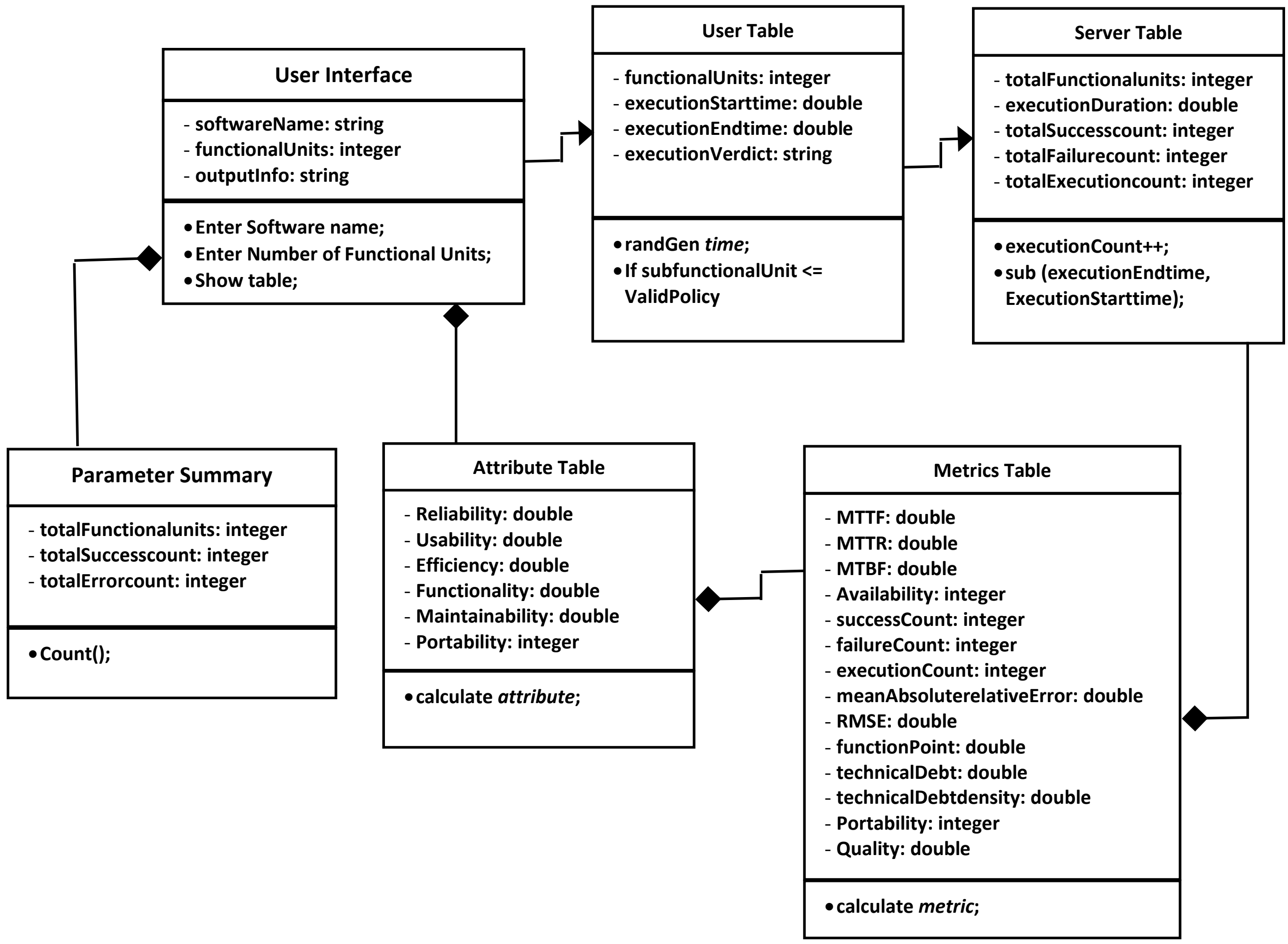


Figure 3.16: Model's Class Diagram.

Figure 3.16 is an illustration of the database contents indicating the classes in which each function belongs. The classes show the formats for the repository contents and what operation makes each module active at any point in time. The classes represent different tables as indicated by their names in the first box of each table frame. The middle section of the frame shows the parameters whose values would be held by each table and finally, the third box shows the operations through which values would be assigned to the parameters.

3.3.16 High Level Design of the System Model

Figure 3.17 gives a detailed architecture of the proposed intelligent evaluation system.

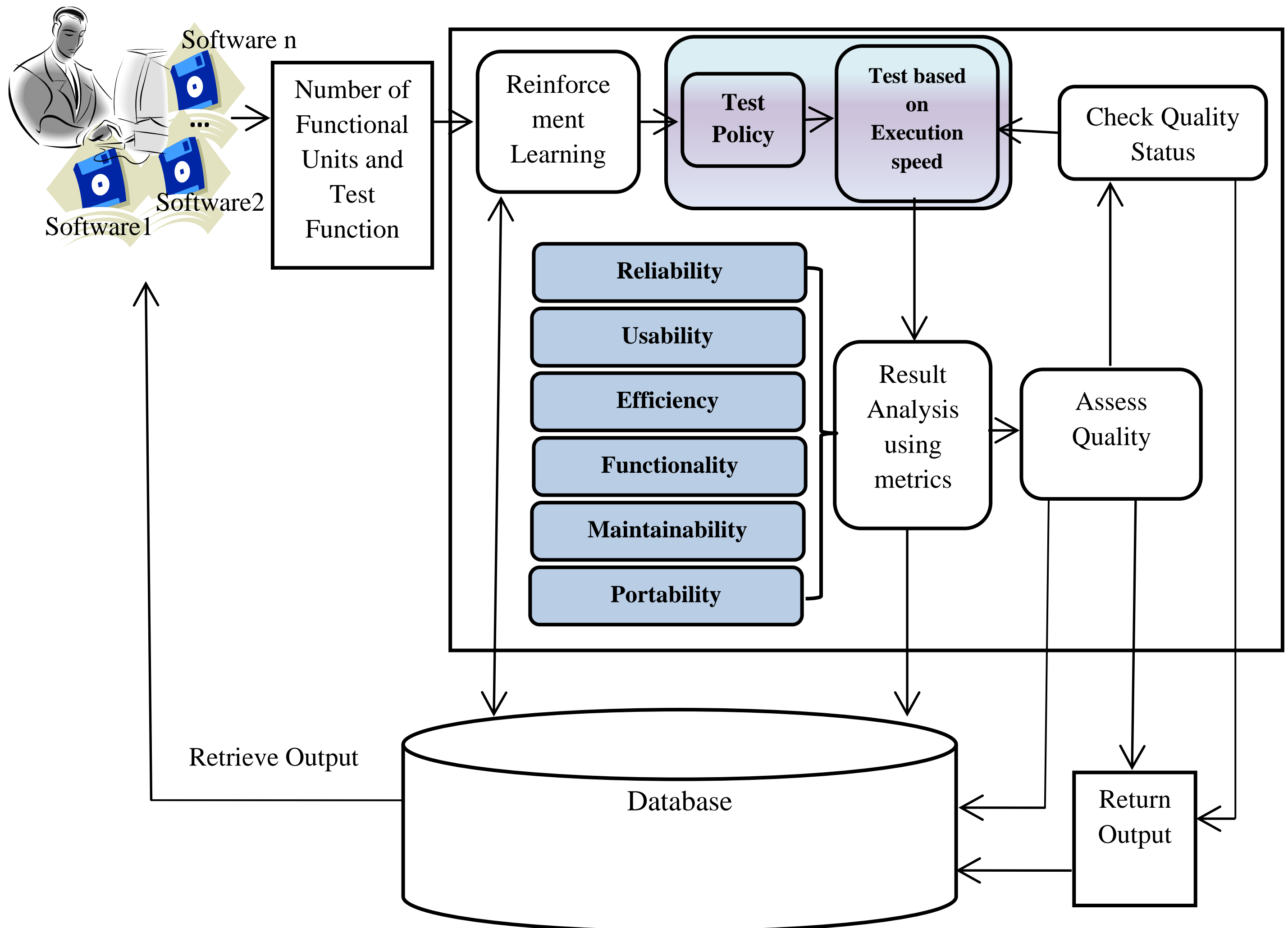


Figure 3.17: Detailed Architecture of the Model.

In Figure 3.17, a user or evaluator feeds the model with some test data. The model's intelligent agent captures the data and determines the strategy best suited for the evaluation using reinforcement learning principles. The agent performs a test using the inputted software data on the basis of the software execution speed and the results are used to measure the quality attributes

(reliability, usability, efficiency, functionality, maintainability, portability) of the software using some built-in metrics. This assessment is repeatedly done on each sub-functional unit of the software in review, with the agent drawing knowledge from previous assessments, until the total number of functional units are exhausted. At the end of the functional units' assessment, their outcomes are used to calculate the quality value of the software undergoing evaluation. The quantified results of the quality attributes and the overall quality are returned as output into a database which is accessible to the user for analysis and decision making.

3.4 Schematic Diagram of the System Model

The model presented in this thesis is translated into a schematic diagram shown in Figure 3.18.

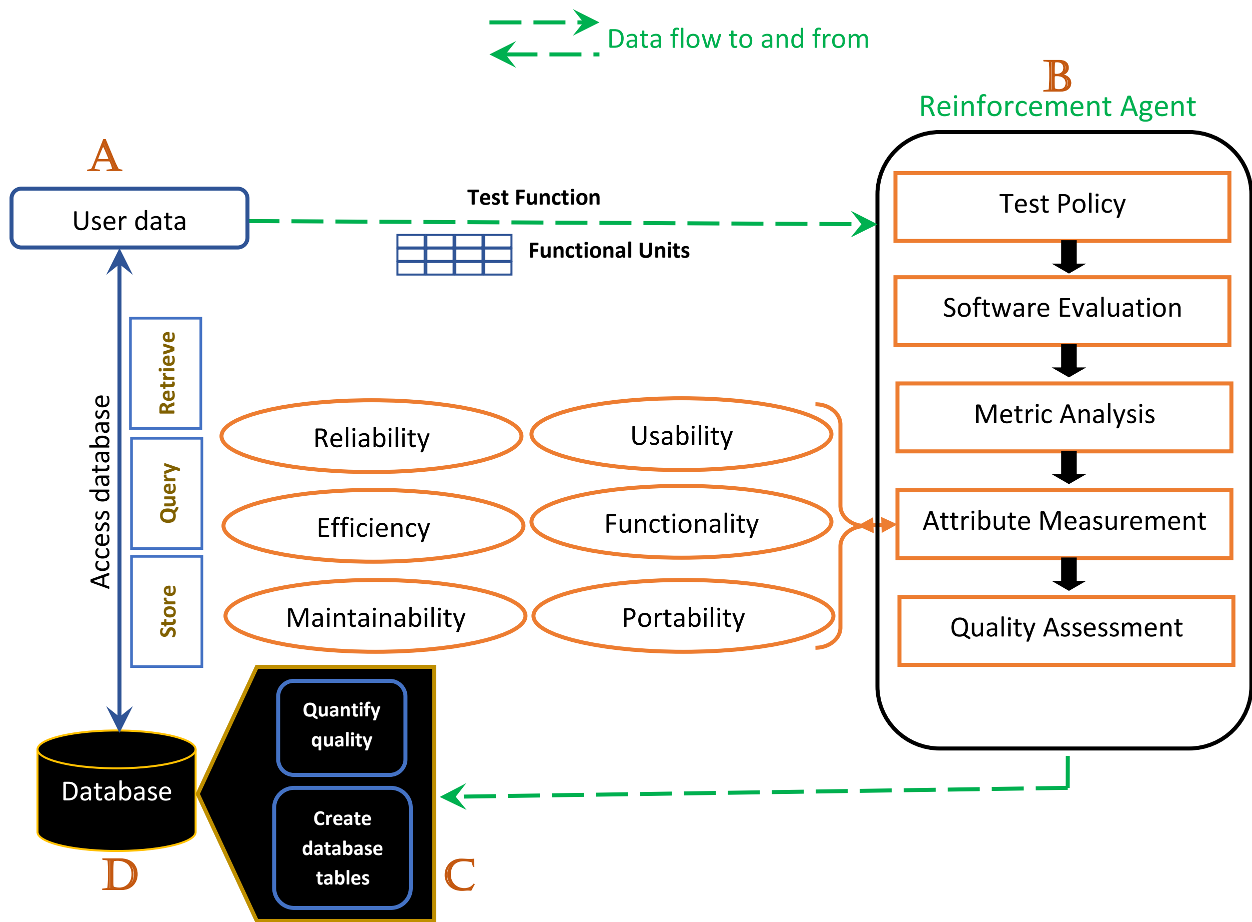


Figure 3.18: Model's Schematic Diagram.

The schematic diagram represents the model as a four-component system which begins from the user and ends with the database repository. The user prepares and transfers the necessary parameters to the reinforcement agent where all measurements and evaluations take place. The quantification unit enables the measurement outcomes to be used in expressing software quality as a single value. The entire system outputs are then stored in the database, which in turn, is accessible to the user for future retrievals.

3.5 Programming Language Used

The intelligent software evaluation model is implemented using Python programming language. Python is a generic high-level programming language with compatibility to different programming methodologies including object-oriented, structured, functional and aspect-oriented programming. Python is a very useful language that supports effective development of both desktop and web-based applications. The dynamism, simple syntax rules and multiple platform compatibility of Python inspired the choice of the language. Most importantly, Python provides machine learning libraries that give the functionality of making a system to learn based on past experiences through stored data.

3.6 Test Data for Model Validation

Test data for the validation of the model in this study were gotten from external sources as primary data. The test-data are the functional information of Oil-palm Management Program (OMP) and Estate CanePro. The information provided were referred to as SW1 and SW2 respectively (see appendix A for details). The information from the two software products were considered relevant since the model only requires functional investigation of any form of software. The data contains the functional units of the two software as well as details of the sub-functional units that make up the software functionality.

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Hardware Specification

The intelligent software evaluation model was designed and simulated on a system with the following specification

Processor:	Intel(R)Core(TM) i3-2350M CPU @2.30GHz 2.30Hz
Installed Memory (RAM):	6.00 GB (5.87 GB usable)
System Type:	64-bit Operating System
Hard disk:	231 GB

4.2 Software Specification

The model was built and simulated on a system with the following software specification.

Operating System:	Windows 10
Web Browser:	Google Chrome
Database server:	MySQL

4.3 Input Interface

The simulation program home page of the model is displayed in figure 4.1.

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

```
    except Error as e:
        print("Error in Mysql connection=",e)
    finally:
        conn.close()
def execute():
    #report,dummy,Execution_Start_Time,Execution_End_Time,SuccessCount,ErrorCount,ExecutionCount,Functional_Unit,Executed_unit,T
    #delta,Beta=Metrics()
    delta,Beta,report,dummy,software,_=decision()
    softwareToSql(software)
    reportToSql(report)
    dummyToSql(dummy)
    deltaToSql(delta)
    BetaToSql(Beta)
Name=Naming()
execute()
print("Done........")


#select * from dummy_hydrogen;  select * from report_hydrogen; select * from delta_hydrogen; select * from beta_hydrogen;
# Drop table delta_hydrogen; --to delete a table
```

DataBase is requesting...Name of software to be stored?:

Figure 4.1: Input Interface on Jupyter Notebook Environment.

Figure 4.1 is a window that shows the interface where the program accepts inputs from the user. At the pressing of "Run" button, the Jupyter Notebook environment prompts an input panel below the program codes in readiness of accepting the necessary data that will trigger the system into action. The process starts by the request for the name that will be used to store the results of the evaluation process (`Name of software to be stored?`). The model further accepts the total number of functional units and thereafter, the number of sub-functions belonging to each of the main functional units.

4.4 Output Interface

Figure 4.2 and 4.3 below shows MySQL Workbench and Command Line (CLI Client) interfaces respectively through which a user gets access to the outputs of the executed program.

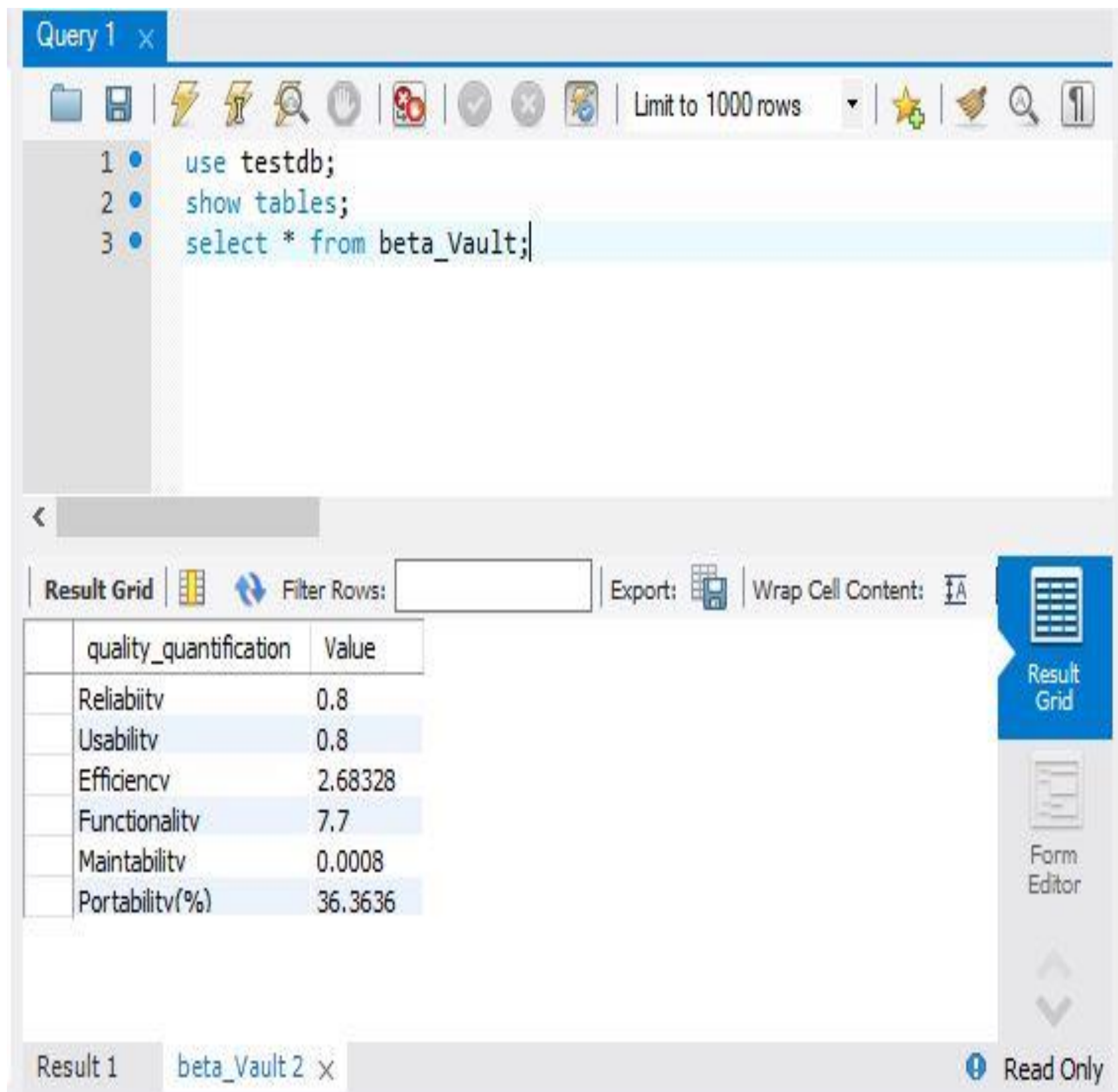


Figure 4.2: MySQL Workbench Output Interface.

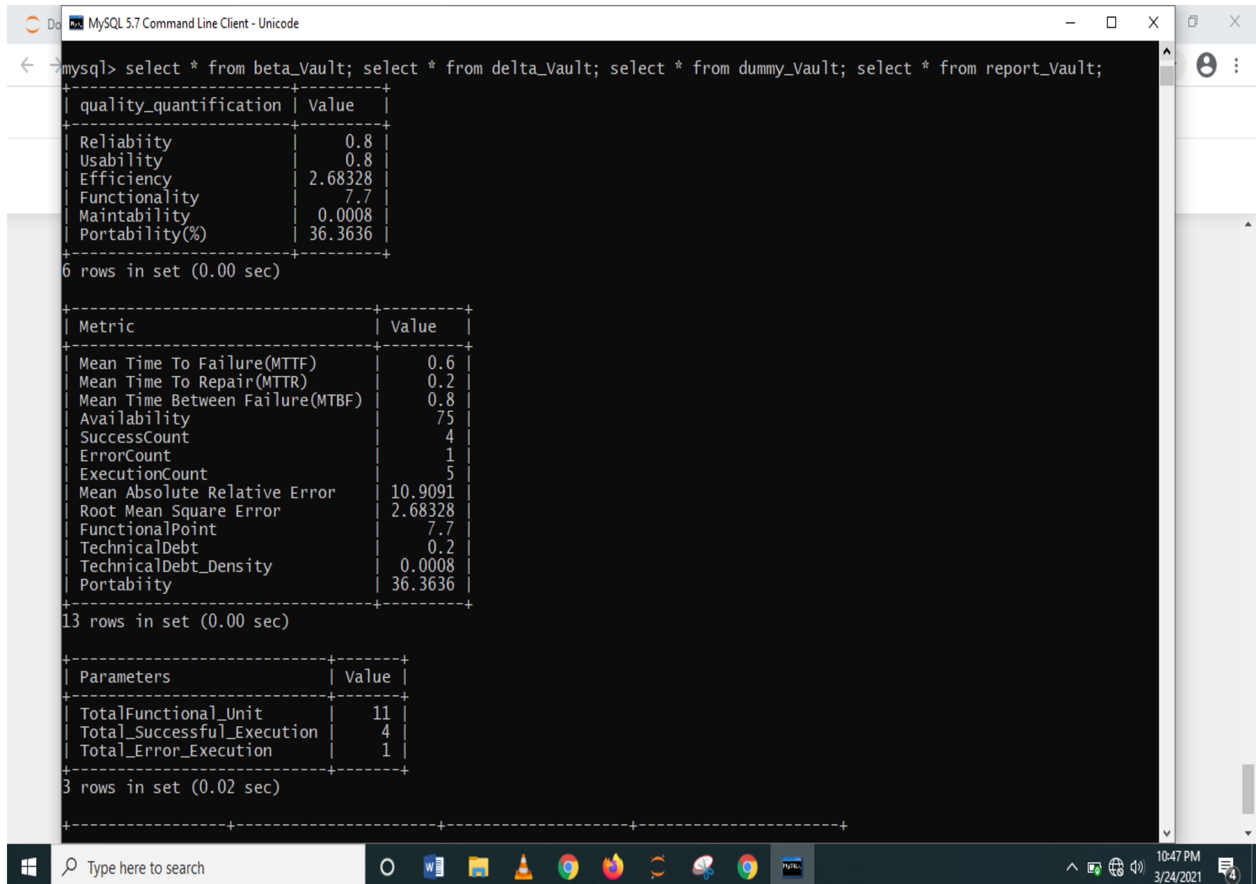


Figure 4.3: MySQL Command Line Output Interface.

The two environments in the respective figures 4.2 and 4.3 provide a platform for querying the database using SQL codes to display the evaluation results which are stored in the repository by default. The query on both environments is done using the same codes (`show tables` and `select * from` *`table_name`*), the difference being that they are displayed in different platforms. The codes enable the database to create views which allow each table to display its contained records when queried accordingly. This confirms the assurance that the database contents are accessible for retrieval and other necessary usages by the user which include decisions for software modification or management analysis.

4.5 Results

Input data gotten from SW1 (Oil Palm Management Program (OMP)), and SW2 (Estate CanePro) respectively as contained in Appendix A yielded the results shown in Tables 4.1 and 4.2.

Table 4.1: Output of Built-in Metrics using SW1 Data.

METRIC	VALUE FOR SW1
Mean Time To Failure (MTTF)	2.2
Mean Time To Repair (MTTR)	0.25
Mean Time Between Failure (MTBF)	2.45
Availability (%)	90
SuccessCount	3
ErrorCount	1
ExecutionCount	4
Mean Absolute Relative Error	10.71
Root Mean Square Error	1.5
FunctionPoint	4.9
TechnicalDebt	0.25
Technical Debt Density	0.001
Portability	43

Table 4.2: Output of Built-in Metrics using SW2 Data.

METRIC	VALUE FOR SW2
Mean Time To Failure (MTTF)	0.6
Mean Time To Repair (MTTR)	0.2
Mean Time Between Failure (MTBF)	0.8
Availability (%)	75
SuccessCount	4
ErrorCount	1
ExecutionCount	5
Mean Absolute Relative Error	10.91
Root Mean Square Error	2.7
FunctionPoint	7.7

TechnicalDebt	0.2
Technical Debt Density	0.0008
Portability	36

The calculated metric values for each of the evaluated software were respectively used to quantify the six (6) quality attributes deployed in this study which are subsequently used to calculate the quality value of the software and their outcomes are given in Table 4.3.

Table 4.3: Quality Quantification.

QUALITY ATTRIBUTE	**SW1 OUTPUT**	**SW2 OUTPUT**	**Legend (> Validity)**
Reliability	0.75	0.78	As value tends to 1
Usability	0.75	0.78	As value tends to 1
Efficiency	1.5	1.3	The smaller, the better
Functionality	4.9	9.1	As value tends to 10
Maintainability (minutes)	0.25	0.2	The smaller, the better
Portability (%)	43	54	As value tends to 100
QUALITY	0.93	1.0	As value tends to 1

4.5.1 Results Analysis

The data gotten from SW1 (OMP) yielded metric values that indicated a relatively short time to recover from any error, a sign that the system can be maintained with less stress. The values also showed that the time it takes for the system to experience an error after a previous one is also reasonable, an indicator that the system contains minimal bugs. The availability of the system is very high as a result of its maintainability and relatively error-free conditions (see Figure 4.4). The ratio of the time it takes the system to fix all detected

faults to the total lines of code that implemented the entire functionality is very small. This shows that the system implementation is in concordance with the principles of extreme programming (XP) adopted in the work. However, the portability of the system tends to be below average which gives an idea that the size of the software is a bit large.

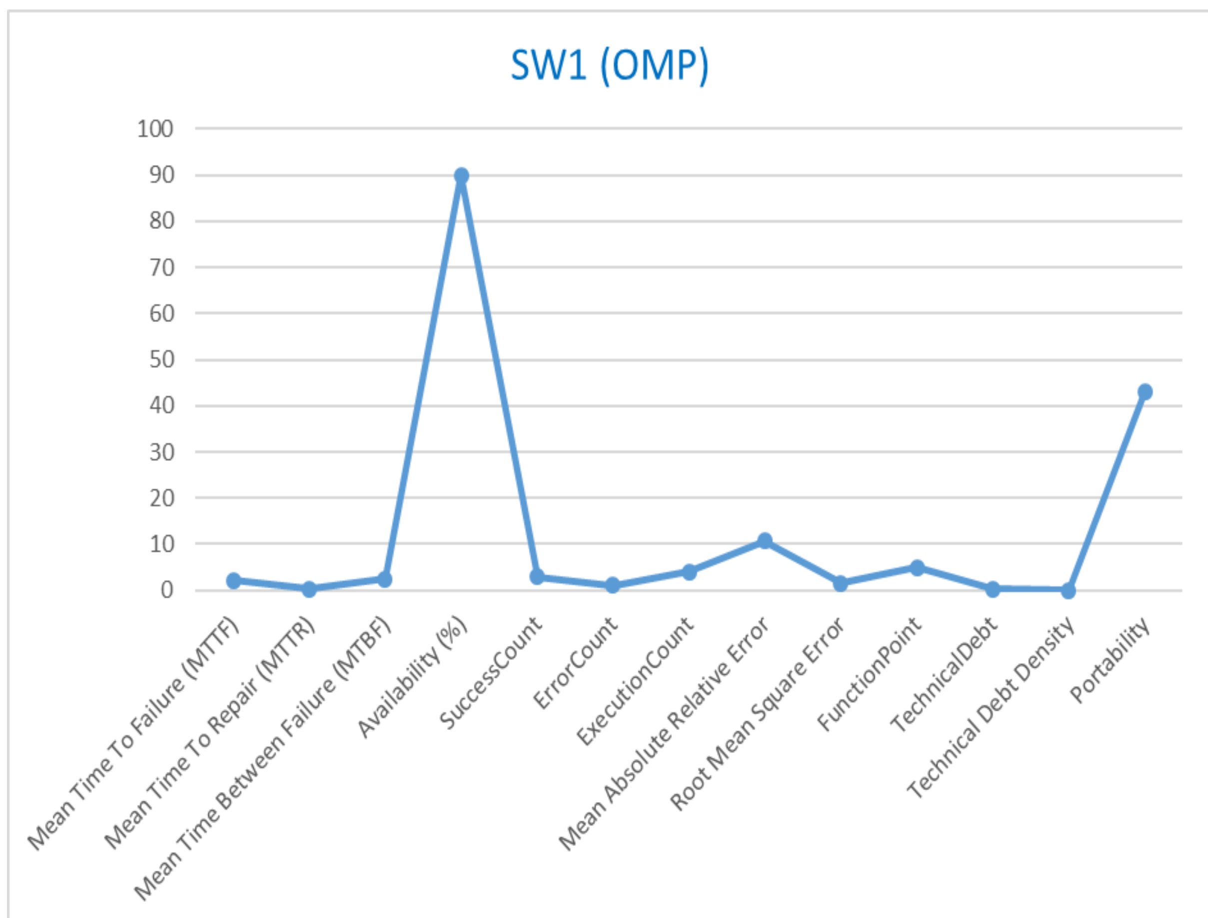


Figure 4.4: Graphical Representation of SW1 Metric Values.

Similarly, SW2 (Estate CanePro) followed the same trend with its predecessor with even a smaller amount of time needed by the system to recover from error. Better still, SW2 has a larger interval of time required by the system to experience a subsequent error. The similar shape of its graph to that of SW1 (as

shown in Figure 4.5) captures the fact that, as the system is more maintainable, the values for system recovery and discovery of fault follow a similar trajectory. The availability of SW2 seems to be less than the former alongside a higher tendency to contain more bugs.

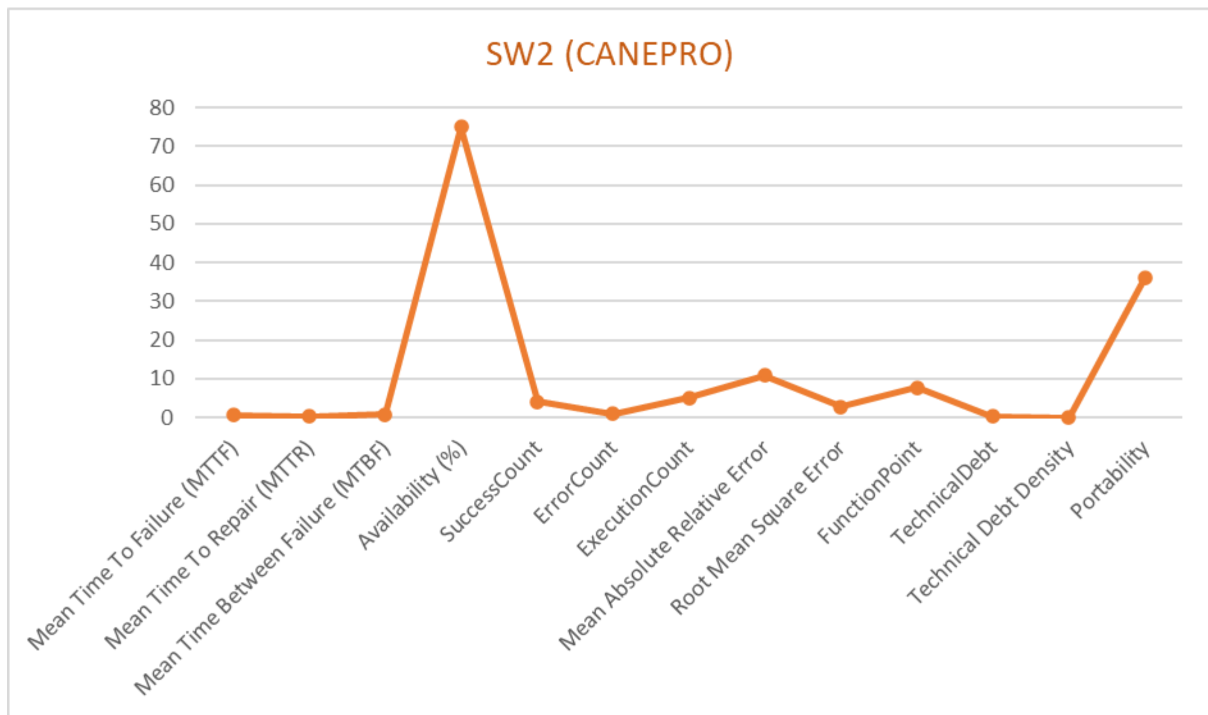


Figure 4.5: Graphical Representation of SW2 Metric Values.

4.6 Discussion of Results

The result of the model's simulation shows that the number of functional units and sub-functional units of a given software can serve as a testing parameter in evaluating the software's quality. The model accepted the required inputs and automatically subjected the test-data into a systematic but intelligent evaluation process, in which it further assesses the software attributes of the software to enable it arrive at a conclusion about the overall quality of the evaluated

software. The model does this via metrication of the software attributes to quantify the attributes in numeric terms. By metrication, we mean the use of software metrics to measure the individual attributes of the software. This process enables the model to present useful information needed by stakeholders about the software in-review which is important for decision making processes. The numeric outcomes of the metrication process are essential for the model's final output reflecting the overall quality of the evaluated software, also in numeric terms. The result of the evaluation will prompt the decision of either to keep running the system as it is, or to implement an improvement measure or even to discard the system in entirety. The model therefore succeeded in quantifying the quality of the evaluated software (using the number of functional units and sub-functional units as the inputs) expressing them as numbers which is a tangible medium for analysis and decision making.

4.6.1 Realization of objective (i)

This work was carried out to achieve the primary objective of designing a software evaluation model that measures the quality of software systems. This objective was successfully achieved with the model architecture shown in Figures 3.3 and 3.9 which was translated into a mathematical model shown in equation (3.1) on page 117. The work identified several program components that combine into an evaluation system which accepts required inputs for the purpose of measuring software quality.

4.6.2 Realization of objective (ii)

The work introduced artificial intelligence into the design's implementation. This was achieved using reinforcement learning which receives software's number of functional units and sub-functional units upon which it generates an action to determine which of the sub-functional units are useful for the evaluation. Reinforcement learning introduced an agent whose function is to condition the system to perform similar actions when placed in an environment that is similar to the initial environment where it had learned from.

The agent returns a value (reward) from the new environment when triggered by an execution such that the independent variables of the environment are matched with test cases generated out of supplying the system with an input dataset. The reinforcement learning agent strategizes on how best the execution could be performed using some self-determined test policy. It figures out the functional units that contains the system information it considers relevant to be capable of revealing the characteristics needed to make a valuable measurement of the quality of the software undergoing evaluation. In other words, the reinforcement agent does not necessarily include all the functional units of the software in the evaluation exercise due to its self-determined reasons. The intelligence applied by the learning agent comes from the `sklearn` library which has the ability to handle evaluation tasks. The system being able to learn from

experience thus, achieved the second objective of the work because it delivers significant information about the evaluated software's performance ability.

4.6.3 Realization of objective (iii)

The work ensured that the designed model was furnished with software metrics whose functions were to measure and quantify the various software attributes in order to achieve a dependable quality assessment. This was realized with metrics related to six (6) software quality attributes namely Reliability, Usability, Efficiency, Functionality, Maintainability, and Portability. The measurement was as a result of the ability drawn from `pandas` and `math` libraries which have the capability of data manipulation. In this way, the third objective of this work was realized successfully.

4.6.4 Realization of objective (iv)

The implementation of the designed system was done with Python and was validated using data from two (2) software systems as contained in Appendix A. The outcomes of the tests (shown in the results above) were analysed and were compared to validate the performance of the software evaluation model presented in this study. The model's performance on the two (2) test-data were observed and plotted. The results of the comparison are pictorially shown in figure 4.6 and are discussed below:

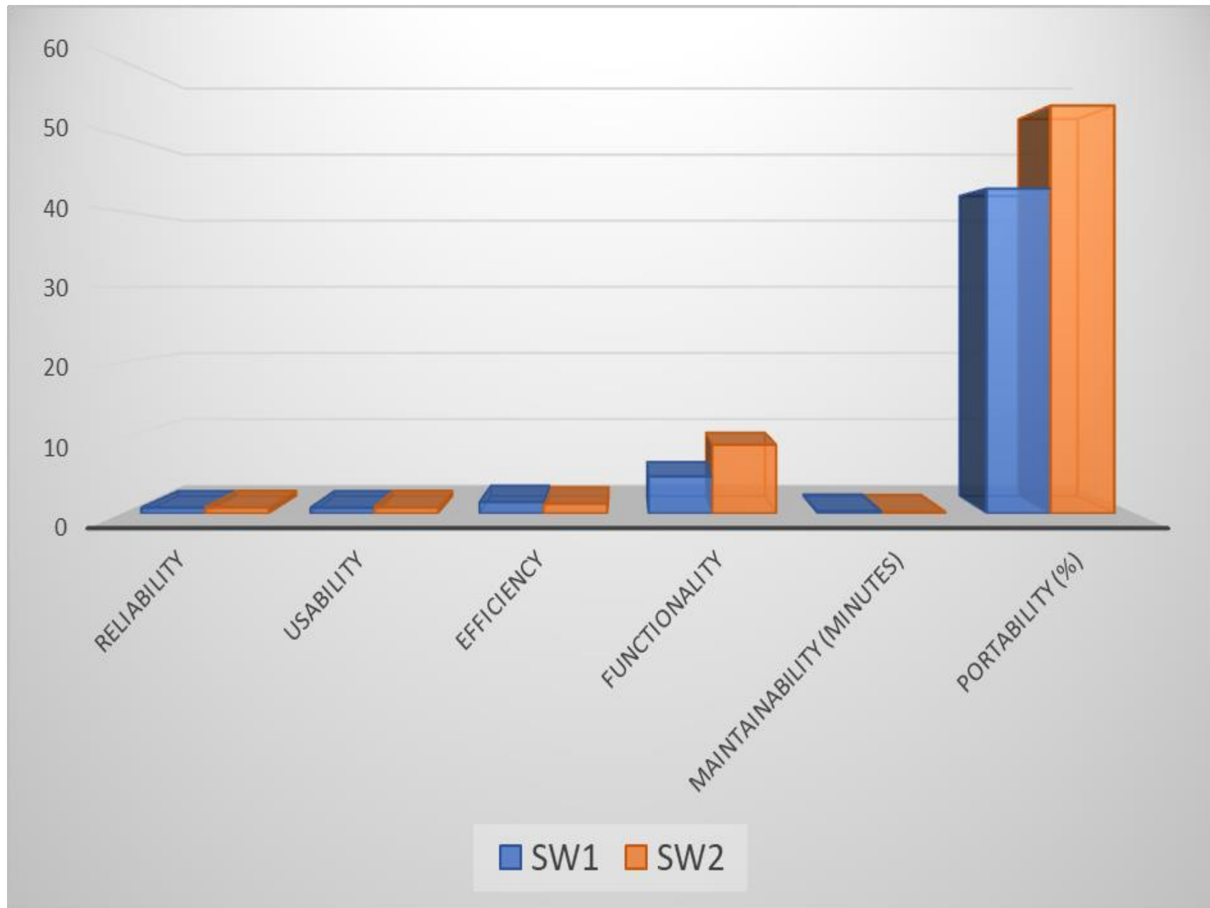


Figure 4.6: Comparative Representation of the Quality Attributes of SW1 and SW2.

Figure 4.6 indicates that firstly, the reliability and usability of a software system is directly proportional to its maintainability. The analogy comes from the mutual indication of the values of the respective metrics of the two software in review towards having a system that is easily maintainable. Secondly, the results show that even though the values of the attributes are reasonable enough, SW2 is more efficient and portable than SW1. This is made clear from SW2's less tendency to contain faults and its high portability value. However, SW2 tends to be of better compliance to its system requirements. The verdict is as a

result of its higher functionality value which tries to prove that its originally intended functions are strictly implemented.

4.7 Comparison of Proposed Model with Existing Systems

The model presented in this thesis is a positive step in the business of software quality assurance. Comparing the system with some existing systems resulted in the following findings:

Hammouri *et al.* (2018) evaluated three machine learning algorithms in their effort to predicting faults in software systems. The model presented in the study for this evaluation purposes utilized historical data to predict potential faults in software systems. The three algorithms (Naïve Bayes (NB), Decision Tree (DT) and Artificial Neural Networks (ANNs)) are supervised forms of machine learning whose performance on evaluation process proved to be effective. Even though bug prediction, as emphasized by the prediction model is a path that leads to quality assurance, this thesis makes an established assertion as regards the quality of a given software in a slight contrast to the provisions of the analysed prediction model. In addition, this thesis eases the difficulty of searching for historical data since the reinforcement learning employed herein makes updates on trial and error.

Pattnaik *et al.* (2018) presented a hybridized prediction model that combines the best features of fuzzy logic and neural network. The model was meant to

capture certain features of interest during development process which enabled the model’s capability to measure a predictive quality of software. However, this thesis makes a categorical establishment regarding software quality measurement, such that, the quality of a given software is represented as a single numeric value.

Pattnaik *et al.* (2019) applied fuzzy logic to propose a pragmatic model that considers the nature of the attributes that contribute to the overall software quality. The model was simulated to evaluate its performance in determining the quality of software systems. The model predicted the software quality factors with the view to quantify them. This thesis moved a step further to quantify the quality of software systems in numeric values with respect to certain attributes as prerequisite measurement standards.

Cowlessur *et al.* (2020) studied various machine learning techniques that have been applied to the prediction of software quality including approaches relating to Artificial Neural Network (ANN), Bayesian Network (BN), Fuzzy Logic (FL), Decision Tree (DT), Support Vector Machine (SVM) and Case-Based Reasoning (CBR). The study discovered that most machine learning-based prediction models presented in literature have the intent of achieving the predictive goal at the early stage of software development with high efficiency as against other techniques. The study however established that more innovative

explorations are yet to be done as regards the application of machine learning in software quality prediction. That is in conformation with the purpose of this thesis geared towards the systematic measurement of software quality using reinforcement learning technique which is yet absent in the myriads of machine learning techniques already applied to software quality prediction.

Omri and Sinz (2021) conducted a survey with focus on the selection and prioritization of test cases as well as the early prediction of faults. The survey gave explanations why deep learning algorithms play conjugal roles in addressing the dichotomy that exist among program semantics and system structure which is capable of predicting the presence of faults. The study further asserted that machine learning's ability to prioritize test cases compels the adaptive model to automate feature generation without altering its meaning and structure. This is beneficial in software maintenance since less important excesses are checked from the onset. This is in line with the methodology adopted in this thesis which has the primary objective of measuring out the quality of software systems in order to give an idea of software competency before their deployment. However, this thesis displayed further functionalities which enables a tangible representation of the software quality for easy assessment purposes.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

5.1 Summary

This study delved into the continuous trend of software measurement and evaluation. The motivation for the research emanated from the complexity and the corresponding reliability importance facing the world of information technology. This is because, having influenced all sorts of life enhancement activities such as transportation, telecommunication, military, industrial process, entertainment offices, aircrafts and business, it becomes absolutely necessary to conduct effective evaluation of software products before their release to potential users. Hence, this thesis researched into the area of software quality assessment with a view to analysing the competence of existing works and to find a way of creating improvement measures in order to ascertain the exact quality of software products through an effective assessment of the software attributes.

In the light of the foregoing, this thesis carried out a review of existing studies on software quality assessment and the application of machine learning algorithms within the niche area. It further proposed a semi-automated model to provide an effective software assessment tool. The model succeeded in fusing software metrics with reinforcement learning in measuring the quality of software systems.

5.2 Contribution to Knowledge

This thesis has presented an intelligent model for the purpose of measuring the quality of software. The model demonstrated the use of software metrics within a reinforcement learning environment to achieve this goal. The innovatory behaviour of the model can be summarized as follows:

i. Quantification of software quality via integration of Reinforcement Learning with automated testing.
ii. Metrics-based measurement of software products in an intelligent framework to ascertain the reliability of such products in order to ensure consistency of quality.
iii. Formulation and implementation of $Quality_{eval} = (\sum_{i=1}^{n} \frac{a_i}{n-1}) * (\frac{\ln(n)}{(n-1)^{1.25}})$ to evaluate software quality.

5.3 Conclusion

Software quality poses much concern to the software developer because a singular deviation from the developer's intention insinuates a total change in the software's intended functionality. However, the outcome of software quality evaluation depends on the evaluation criteria approved by the user community. This is the only time the value of the evaluation would gain credibility within the user's circle. That is to say that software quality measurement and evaluation should be aimed at providing confidence to the user about the

correctness of the software product. This thesis keyed into customer satisfaction by presenting an intelligent model that contains built-in metrics to measure the quality of software products using the number of functional units. The model further quantifies six (6) software quality attributes - Reliability, Usability, Efficiency, Functionality, Maintainability, and Portability as measurement standards for quality evaluation. The attribute quantities are then used to evaluate the software quality via a formulated mathematical model (i.e., equation 3.1). The model's intelligence comes from Reinforcement Learning principles which creates room for optimal decision making to ascertain which functional units give the required quality measurement. This thesis has therefore demonstrated that metrics-based intelligent software evaluation tool gives efficient value of the quality of software systems. It also proved that users' trust on software performance can be achieved using the model.

5.4 Recommendations

In view of the outcomes of this study, we make some recommendations to both readers and researchers. Firstly, we recommend the model presented in this thesis to software developers and researchers involved in test-driven development (TDD). This is because it is of utmost importance to ascertain the competence of software before its release for use, to avert disastrous consequences that may arise out of undiscovered errors which could be avoidable or correctible through software evaluation. Secondly, we recommend

that further research could be carried out in the direction of going deeper with the software attributes involved in our model by training the agent with their various sub-characteristics alongside other inputs as parameters for software measurement. Thirdly, research can also be directed towards introducing some other software attributes to replace the ones already used in this model and comparing their results to ascertain the group of attributes that gives a better software quality measurement.

REFERENCES

AcqNotes. (2020). Evaluation Criteria. 1-3. Retrieved from https://acqnotes.com/acqnote/task/evaluation-criteria on 09/07/2021.

Akbari, M., & Rajabi, M. A. (2013). Evaluation of Desktop Free/Open Source GIS Software Based on Functional and Non-Functional Capabilities, Department of Geomatics Engineering, University of Tehran, Technical Gazette 20, 5(2013), 755-764.

Alashqar, A. M., Elfetouh, A. A., & El-Bakry, H. M. (2015). ISO 9126 Based Software Quality Evaluation Using Choquet Integral. International Journal of Software Engineering & Applications (IJSEA), 6(1), 111-121.

Alrawashdeh, T. A., Muhairat M., & Althunibat, A. (2013). Evaluating the Quality of Software in ERP Systems Using the ISO 9126 Model. International Journal of Ambient Systems and Applications (IJASA) 1(1), 1-9.

Altexsoft. (2019). Quality Assurance, Quality Control and Testing – The Basics of Software Quality Management. Altexsoft White Paper, 1-25.

Alvaro, A., Almeida, E.S., & Meira, S. R. L. (2010). A Software Component Quality Framework, *ACM SIGSOFT SEN 35, 1* (Mar. 2010), 1-4.

Andrzejak, A., & Silva, L. (2008). Using machine learning for non-intrusive modeling and prediction of software aging. Network Operations and Management Symposium, NOMS 2008. IEEE, 25–32.

Ayodele, T. O. (2010). Types of Machine Learning Algorithms, New Advances in Machine Learning, Yagang Zhang (Ed.). InTechOpen Publishers, Rijeka, Croatia, 19-48.

Ayon, D. (2016). Machine Learning Algorithms: A Review, International Journal of Computer Science and Information Technologies, 7(3), 1174-1179.

Ayyildiz, T. E., & Erkal, B. (2019). The Effect of Object Oriented Metrics on Software Bug Prediction. Journal of Information Systems and Management Research, 1–8.

Bajpai, P., Chaturvedi, R., & Singh, A. (2019). Conjecture of Scholars Academic Performance using Machine Learning Techniques. International Conference on Cutting-edge Technologies in Engineering (ICon-CuTE) Shri Ramswaroop Memorial University, Barabanki, India, 141–146.

Bertoa, M. F., & Vallecillo, A. (2006). Usability metrics for software components, Dpto. Lenguajes y Ciencias de la Computación. Universidad de Málaga, 1-10.

Bevan, N., & Macleod, M. (1994). Usability measurement in context, Behavior and Information Technology, 13: 132–145.

Bindal, D. (2013). A Review of Markov Model for Estimating Software Reliability, International Journal of Advanced Research in Computer Science and Software Engineering, Kurukshetra University, Haryana, India, 3(6):426-433.

Bo-Zhou, B., Okamura, H., & Dohi, T. (2013). Enhancing Performance of Random Testing Through Markov Chain Monte Carlo Methods, IEEE Transactions on Computers, University of California Riverside, 62(1):1-4.

Brun, Y., & Ernst, M. D. (2004). Finding latent code errors via machine learning over program executions. Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, 480–490.

Chappell, D. (2018). The Three Aspects of Software Quality: Functional, Structural, and Process. Microsoft Corporation White Paper, 1-6. http://www.davidchappell.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf

Chen, N., Hoi, S. C. H., & Xio, X. (2011). Software Process Evaluation: A Machine Learning Approach, 26th IEEE/ACM International Conference on Automated Software Engineering, Research Collection of Information Systems, 333-342.

Chren, S., Macak, M., Rossi, B., & Buhnova, B. (2022). Evaluating Code Improvements in Software Quality Course Projects. In the International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022), June 13-15, 2022, Gothenburg, Sweden. ACM, New York, NY, USA, 10p. https://doi.org/10.1145/3530019.3530036

Ciaschini, V., Canaparo, M., Ronchieri, E., & Salomoni, D. (2014). Evaluating Predictive Models of Software Quality, 20th International Conference on Computing in High Energy and Nuclear Physics, Journal of Physics: Conference Series 513 (2014) 052030, 1-7.

Cingil, T., & Sozer, H. (2022). Black-box Test Case Selection by Relating Code Changes with Previously Fixed Defects. In The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE

2022), June 13–15, 2022, Gothenburg, Sweden. ACM, New York, NY, USA, 10p. https://doi.org/10.1145/3530019.3530023

CISQ Workgroups for Reliability, Performance Efficiency, Security and Maintainability. (2012). CISQ Specifications for Automated Quality Characteristic Measures, CISQ-TR-2012-01, Consortium for IT Software Quality, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Cowlessur, S. K., Pattnaik, S., & Pattanayak, B. K. (2020). A Review of Machine Learning Techniques for Software Quality Prediction: In Advanced Computing and Intelligent Engineering, Springer Nature Singapore Pte Ltd, 537-549.

Dean, J. C., & Vigder, M. R. (2000). COTS Software Evaluation Techniques, RTO MP-48, Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS", Brussels, Belgium, 1-6.

Farooq, S. U., Quadri, S. M. K., Ahmad, N. (2011). Software Measurements and Metrics: Role in Effective Software Testing. International Journal of Engineering Science and Technology, 3(1), 671-680.

Fenton, N., & Bieman, J. (2014). Software metrics: a rigorous and practical approach. CRC press.

Figueiredo, E. (2014). Evaluation in Software Engineering. Retrieved from http://www.dcc.ufmg.br/~figueiredo on 20/04/2018, 1-13.

Ganney, P. S., Taktak, A., Long, D., & Axell, R. G. (2020). Clinical Engineering: A Handbook for Clinical and Biomedical Engineers, Second Edition. Academic Press, Elsevier Ltd, 1-530.

Gediga, G., Hamborg, K., & Düntsch, I. (2015). Evaluation of Software Systems, Institut für Evaluation und Marktanalysen Brinkstr. 19, 49143 Jeggen, Germany, 1-43.

Geeksforgeeks. (2020). Software Measurement and Metrics. Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305, 1-3. Retrieved from https://www-geeksforgeeks-org.cdn.ampproject.org/v/s/www.geeksforgeeks.org/software-measurement-and-metrics/ on 09/07/2021.

Ghandorh, H., Noorwali, A., Nassif, A. B., & Eagleson, L. F. C. R. (2020). A Systematic Literature Review for Software Portability Measurement: Preliminary Results. ICSCA, Langkawi, Malaysia, 152-157.

Gillies, A. (2011). Software Quality: Theory and Management. Retrieved from https://www.lulu.com on 7/10/2021.

Hamborg, K., Vehse, B., & Bludau,H. (2004). Questionnaire Based Usability Evaluation of Hospital Information Systems. Electronic Journal of Information Systems Evaluation 7(1), 21-30.

Hammouri, A., Hammad, M., Alnabhan, M., & Alsarayrah, F. (2018). Software Bug Prediction using Machine Learning Approach. International Journal of Advanced Computer Science and Applications, 9(2), 78-83.

Harmon, S. Y., & Metz, M. L. (2014). A Quantitative Approach to Software Quality Evaluation**,** Innovative Management Concepts, 2014 Software Technology Conference, 1-23.

Harley, N., & Spaven, F. (2017). Software intelligence – The secret to building flawless web and mobile apps. Raygun Limited,1-49.

Hassan, A. E., & Xie, T. (2010). Software Intelligence: The Future of Mining Software Engineering Data. FoSER November 7–8, 2010, Santa Fe, New Mexico, USA, ACM 978-1-4503-0427-6/10/11, 1-6.

Heini, O. (2007). Performance measurement: Designing a Generic Measure and Performance Indicator Model. Information Systems Department, University of Geneva, 1-112.

Hlomani, H., & Stacey, D. (2014). Approaches, methods, metrics, measures, and subjectivity in ontology evaluation: A survey, School of Computer Science, University of Guelph, Ontario Canada, 1-11.

Institute of Electrical and Electronics Engineers. (1990). IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, 1-84. Doi: 10.1109/IEEESTD.1990.101064

Institute of Electrical and Electronics Engineers. (2006). IEEE Standard for Software Maintenance. IEEE Std 14764-2006, 1-56. Doi: 10.1109/IEEESTD 2006.88278

Isitan, K. K. (2011). An Approach to Establish a Software Reliability Model for Different Free and Open Source Software Development Paradigms, Tampere University, Tampere, 6-8.

International Organization for Standardization. (2006). Ergonomic Requirements for Office Work with Visual Display Terminals – Part 10: Usability: Definitions and Concepts. ISO 9241-110:2006, 1-11.

International Organization for Standardization. (2018). Ergonomics of Human System Interaction – Part 11: Usability: Definitions and Concepts. ISO 9241-11:2018(en).

International Organization for Standardization/International Electrotechnical Commission. (2014). Software Engineering: Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. ISO/IEC 25000:2014, 1-41.

International Organization for Standardization/International Electrotechnical Commission. (2011). Software Engineering- Product quality-part 1: Quality model. ISO/IEC 25010:2011.

International Organization for Standardization/International Electrotechnical Commission. (2016). Software Engineering- Product quality-part 2: External Metrics. ISO/IEC 25023:2016.

International Organization for Standardization/International Electrotechnical Commission. (2016). Software Engineering- Product quality-part 3: Internal Metrics. ISO/IEC 25023:2016.

International Organization for Standardization/International Electrotechnical Commission. (2016). Software Engineering- Product quality-part 4: Quality in Use metrics. ISO/IEC 25022:2016.

Jackson, M., Crouch, S., & Baxter, R. (2011). Software Evaluation: Criteria-based Assessment. Software Sustainability Institute, 1-13. Retrieved from http://www.sourceforge.net/projects/ogsa-dai on 21/04/2018.

Jadha, A. S., & Sonar, R. M. (2009). Evaluating and selecting software packages: A review. Information and Software Technology 51, 555–563.

Jeanrenaud, A., & Romanazzi, P. (1994). Software product evaluation metrics: a methodological approach, Transactions on Information and Communications Technologies, WIT Press, ISSN 1743-3517, 9, 59-69.

Jung, H., Kim, S., & Chung, C. (2004). Measuring Software Product Quality: A Survey of ISO/IEC 9126, IEEE Computer Society, 88-92.

Kaelbing, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research, 4, 237-285.

Kapoor, P., Arora, D., & Kumar, A. (2018). Investigating Implications of Metric Based Predictive Data Mining Approaches towards Software Fault

Predictions. International Journal of Engineering &Technology, 7 (3.12), 427-433.

Kassie, N. B., & Singh, J. (2020). A study on software quality factors and metrics to enhance software quality assurance. International Journal of Productivity and Quality Management, 1-21.

Kaur, G., & Bahl, K. (2014). Software Reliability, Metrics, Reliability Improvement Using Agile Process. International Journal of Innovative Science, Engineering & Technology, 1(3), 143-147.

Kim, M., Lee, S., Lim, J., Choi, J., & Kang, S. G. (2020). Unexpected Collision Avoidance Driving Strategy Using Deep Reinforcement Learning. 10.1109/ACCESS.2020.2967509, IEEE Access, 1–10.

Kitchenham, B., & Pfleeger, S. L. (1996). Software Quality: The Elusive Target. IEEE Software 13(1), 12-21.

Kopyltsov, A. V. (2020). Selection of metrics in software quality evaluation. Journal of Physics: Conference Series, ICMSIT 2020, IOP Publishing Ltd, 1–6.

Lai, X., & Brinkkemper, S. (2007). Concepts of Product Software: Paving the Road for Urgently Needed Research, Technical report, Institute of Information and Computing Sciences, Utrecht University, The Netherlands. European Journal of Information Systems 16, 531–541.

Lenhard, J., & Wirtz, G. (2015). Measuring the Portability of Executable Service-Oriented Processes. IEEE International Conference EDOC, 17(2015), 2-4.

Li, L., Lessmann, S., & Baesens, B. (2019). Evaluating software defect prediction performance: an updated benchmarking study. Kent Business School, University of Kent, United Kingdom, 1–21.

Li, Y., Lee, S-Y., Wotawa, F., & Wong, W. E. (2019). Using Tri-Relation Networks for Effective Software Fault-Proneness Prediction. 10.1109/ACCESS.2019.2916615, IEEE Access, 1–15.

Lochmann, K. (2013). Defining and Assessing Software Quality by Quality Models. Institut für Informatik der Technischen Universität München, Germany, 11–173.

Lyras, D. P., Panagiotakopoulos, T. C., Kotinas, I. K., Panagiotakopoulos, C. T., Sgarbas, K. N., & Lymberopoulos, D. K. (2014). Educational Software

Evaluation: A Study from an Educational Data Mining Perspective. The International Journal of Multimedia & Its Applications (IJMA) 6(3), 1-20.

Lysne, O. (2018). Software Quality and Quality Management. The Huawei and Snowden Questions, Simula SpringerBriefs on Computing 4, 87-98. https://doi.org/10.1007/978-3-319-74950-1_10

Malathi, S., & Sridhar, S. (2012). Analysis of size metrics and effort performance criterion in software cost estimation. Indian Journal of Computer Science and Engineering, 3(1), 24-31.

Mansoor, A., Streitferdt, D., & Fubi, F. (2015). Fuzzy Based Evaluation of Software Quality Using Quality Models and Goal Models, (IJACSA) International Journal of Advanced Computer Science and Applications, 6(9), 265-273.

Miguel, J. P., Mauricio, D., & Rodríguez, G. (2014). A Review of Software Quality Models for the Evaluation of Software Products. International Journal of Software Engineering & Applications (IJSEA), 5(6), 31-53.

Misra, S., Akman, I., & Colomo-Palacios, R. (2013). A Framework for Evaluation and Validation of Software Complexity Measures. Department of Computer Engineering, Atilim University, Ankara, Turkey, 1-27.

MITRE Corporation. (2021). Test and Evaluation. Retrieved from https://www.mitre.org/publications/systems-engineering-guide/se-lifecycle-building-blocks/test-and-evaluation on 08/10/2021.

Mohssen M., Muhammad, B. K., & Eihab, B. M. B. (2017). Machine Learning: Algorithms and Applications. CRC Press by Taylor & Francis Group, Boca Raton, 1-14.

Moreno, V., Génova, G., Parra, E., & Fraga, A. (2020). Application of machine learning techniques to the flexible assessment and improvement of requirements quality. Software Quality Journal, 1-30.

Moser, S. (1996). Measurement and Estimation of Software and Software Processes. University of Bern, Switzerland, 1–152. Retrieved from https://scg.unibe.ch/archive/phd/moser-phd.pdf on 21/04/2018.

Munaiseche, C. P. C., & Liando, O. E. S. (2016). Evaluation of expert system application based on usability aspects, International Conference on Innovation in Engineering and Vocational Education, IOP Conf. Series: Materials Science and Engineering 128 (2016) 012001, 1-10.

Munson, J. C. (2003). Software Engineering Measurement. CRC Press, Inc., Boca Raton, FL, USA.

Nakai, H., Tsuda, N., Honda, K., Washizaki, H., & Fukazawa, Y. (2016). A SQuaRE-based Software Quality Evaluation Framework and its Case Study. IEEE International Conference on Software Quality, Reliability & Security, 1-4.

Nicchiotti, G., Fromaigeat, L., & Etienne, L. (2016). Machine Learning Strategy for Fault Classification Using Only Nominal Data. European Conference of the Prognostics and Health Management Society, 1–9.

Nwandu, I. C., & Asagba, P. O. (2017). Automated Software Testing For Reliable System Development. The International Journal of Science and Technoledge, 5(12), 44-49.

Oberscheven, F. M. (2014). Software Quality Assessment in an Agile Environment, Faculty of Science Radboud University, Nijmegen, 1-82.

Omri, S., Sinz, C., & Montag, P. (2019). An Enhanced Fault Prediction Model for Embedded Software based on Code Churn, Complexity Metrics, and Static Analysis Results. Karlsruhe Institute of Technology, Germany, 1–7. Retrieved from https://www.researchgate.net/publication/338554777 on 27/06/2020.

Omri, S., & Sinz, C. (2021). Machine Learning Techniques for Software Quality Assurance: A Survey. Institute of Theoretical Informatics, Karlsruhe Institute of Technology, 1-10.

Oppermann, R., & Reiterer, H. (1997). Software evaluation using the 9241 evaluator. Behaviour & Information Technology; 16 (1997), 4-5, 232-245.

Pattnaik, S., Pattanayak, B. K., & Patnaik, S. (2018). Prediction of Software Quality Using Neuro-Fuzzy Model. International Journal of Intelligent Enterprise, 5(3), 292-307.

Pattnaik, S., Pattanayak, B. K., & Patnaik, S. (2019). Software Quality Prediction Using Fuzzy Logic Technique. International Journal of Information Systems in the Service Sector, 11(2), 51-65.

Pfleeger, S. L. (1998), Software Engineering, Theory and Practice, Prentice-Hall, Inc.

Pfleeger, S. L., & Atlee, J. M. (2010), Software Engineering, Theory and Practice, 4th Edition. Pearson.

Pfleeger, S. L., Fenton, N., & Page, S. (1994). Evaluating Software Engineering Standards, *Computer*, 27(9), 71–79.

Rana, R., & Staron, M. (2015). Machine Learning Approach for Quality Assessment and prediction in Large Software Organisations, University of Gothenburg, Sweden, 1-5. Retrieved from https://www.researchgate.net on 14/05/2019.

Rana, Z. A., Awais, M. M., & Shamail, S. (2010). Nomenclature Unification of Software Product Measures. IET Software, Lahore University of Management Sciences (LUMS), Sector U, DHA Lahore Cantt., Lahore 54792, Pakistan, 1–25.

Schulmeyer, G. G. (2008). Handbook of Software Quality Assurance (Fourth Edition). ARTECH HOUSE, INC. 685 Canton Street Norwood, MA 02062, 1-13.

Seffah, A., Donyaee, M., Kline, R. B., & Padda, H. K. (2006). Usability measurement and metrics: A consolidated model, Software Qual J, 14: 159–178.

Shafiq, S., Mashkoor, A., Mayr-Dorn, C., & Egyed, A. (2020). Machine Learning for Software Engineering: A Systematic Mapping. Johannes Kepler University, Linz, Austria, 1-20. Retrieved from https://www.researchgate.net/publication/341699387 on 13/07/2020.

Sherief, N., Jiang, N., Hosseini, M., Phalp, K., & Ali, R. (2014). Crowdsourcing Software Evaluation, Faculty of Science and Technology, Bournemouth University, UK, 1-4.

Soliman, T. H. A., El-Swesy, A., & Ahmed, S. H. (2010). Utilizing CK Metrics Suite to UML Models: A Case Study of Microarray MIDAS Software. The 7th International Conference on INFOrmatics and Systems, 8–13.

Sommerville, I. (2016). Software Engineering: Tenth Edition. Pearson Education Limited, ISBN 978-0-13-394303-0

Stoilova, K., & Stoilov, T. (2005). Software Evaluation Approach. Institute of Computer and Communication Systems – Bulgarian Academy of Sciences, 1-11.

Sutton, R. S. (1992). Introduction: The Challenge of Reinforcement Learning, Kluwer Academic Publishers, Boston, Machine Learning, 8, 225-227.

Tekinerdogan, B., Ali, N., Grundy, J., Mistrik, I. & Soley, R. (2016). Software Quality Assurance in Large Scale and Complex Software-intensive

Systems: Chapter one. Elsevier Inc., 1-17. doi: https://doi.org/10.1016/C2014-0-01795-9

Tsantekidis, A., Passalis, N., Toufa, A., Saitas-Zarkias, K., Chairistanidis, S., & Tefas, A. (2020). Price Trailing for Financial Trading using Deep Reinforcement Learning. IEEE Transactions on Neural Networks and Learning Systems, 1-10.

Tutorial Point. (2020). Software Measurement. Retrieved from https://www.tutorialspoint.com/software_quality_management/software_quality_management_measurement.htm on 8/7/2021.

Uddin, F. (2022). Software Quality Assurance, Testing, and Reliability. Retrieved from https://www.linledin.com/pulse/software-quality-assurance-testing-reliability-fahim-uddin on 21/11/2022.

Uqaili, I., & Ahsan, S. (2019). Machine Learning Based Prediction of Complex Bugs in Source Code. International Arab Journal of Information Technology, 1–13.

Viljanen, J. (2015). Measuring software maintainability, School of Science, Aalto University, Helsinki, 19-21.

Wang, R. T. (2014). Reliability Evaluation Techniques, Department of Statistics, Tunghai University, Taichung, Taiwan, 31-69.

Yahaya, J., Deraman, A., Kamaruddin, S. S., & Ahmad, R. (2011). Development of a Dynamic and Intelligent Software Quality Model. ICIEIS 2011, Part I, CCIS 251, Springer-Verlag Berlin Heidelberg, 537–550.

Zhang, D. (2002). Applying Machine Learning Algorithms in Software Development. Department of Computer Science California State University, 1–11.

Zhang, D., & Tsai, J. J. P. (2003). Machine Learning and Software Engineering. Software Quality Journal, 11(2), Kluwer Academic Publishers, Netherlands, 87–119.

APPENDIX A

VALIDATION DATA

a. SW1: Oil-palm Management Program (OMP)

OMP is a software developed by Agrisoft Systems which provides solutions for precision oil-palm estate management and agronomy. It contains a main application alongside a number of add-in programs that offer solutions for specific aspects of oil-palm management practices.

SW1 TEST DATA

S/N	Functional Unit	Title of Sub-functional Units	No. of Sub-functional Units
1	Main Menu	Overviews, Planting Information, Production & Harvesting, Fertilization & Nutrients, Vegetative Growth, Field Upkeep, Weather & Climate, Pests & Diseases, Site CHARACTERISTICS, Spatial Unit Definitions, Settings & Tools	11
2	OMP Add-Ins	CLA, Black Bunch Count (BBC), Crop Budgeting (CB), Field Survey (FS), PU, Fertilizer Planning (FP), MT	7
3	OMP-Harvest Round Recording (HRR)	Daily Overview Report, Yield Analysis, Data Entry, System Setting, Yield Reports, Harvest Process Analysis, Harvest Process Reports, Chart, Data Import, Data Export	10
4	OMP-GIS	Web link	1
5	Sort & Filter	Sort A – Z, Sort Z – A, Filter, Clear Filter	4
6	External data	Import from Excel, Raw Data to Excel	2
7	Application (About OMP)	Web link	1

b. SW2: Estate CanePro

Estate CanePro is a multifaceted management tool tailored to large enterprise agronomical estates. It was originally designed for sugarcane farm management.

However, its design that suits multiple users across multiple departments invigorated its application to other forms of farm estate management.

SW2 TEST DATA

S/N	Functional Unit	Title of Sub-functional Units	No. of Sub-functional Units
1.	Field Records	Historical data, Current data, Future forecast, Data analysis, Import data	5
2.	Agronomy Data	Pest control, Disease control, Soil analysis, Leaf analysis	4
3.	Human Resources & Payroll	Employee ID, Employee daily activities, Employee daily productivity, Export data, Generate payslip	5
4.	Dynamic Harvest Planner	Field name, Mill demand, Field harvest order, Harvest planner	4
5.	Resource Planning & Budgeting	Resource planner, Financial Budgeting, Operational planner	3
6.	CanePro Mobile	Capture field data, View field data, Corrective Actions	3
7.	Stores & Materials	Store items, Material items, Material application on field	3
8.	Yield Forecasting & Benchmarking	Crop growth, Output forecast, Irrigation scheduling, Field performance benchmark	4
9.	Replant Planning	Replant planner, Plant variety, Replant harvest date, Replant capacity, Potential Replant period	5
10.	Remote Cane Management	RCM weblink	1
11.	Fleet Management	Vehicle fueling, Vehicle utilization, Service schedule, Workshop job	4
12.	Dashboard Reporting	Graphs, Maps, Custom thematics, Filter	4
13.	Irrigation Scheduling & Meteorology	Irrigation scheduling, Irrigation management, Water supply, Water budget, Power budget, Water analysis	6

APPENDIX B

PROGRAM CODE

```
def initialization():
    params={}
    import random, numpy as np
    np.random.seed(3)

    params['init_state']=np.abs((np.random.randn(100,1)*2.
    7).round())
    params['final_state']=np.abs((np.random.randn(100,1)*6
    ).round())

    params['state_transition']=np.abs((np.random.randn(100
    ,1)*5).round())

    params['Execution_Start_Time']=np.abs((np.random.randn
    (100,1)*2).round(1))

    params['adder']=np.abs((np.random.randn(100,1)*1.5).ro
    und(1))

    params['Execution_End_Time']=(params['Execution_Start_
    Time']+params['adder']).round(1)

    params['Calculate_TimeInterval']=(np.abs((params['Exec
    ution_Start_Time']+params['Execution_End_Time']))/2)
```

```
    Valid_Execution_Speed=random.randint(2,3)
    floated=np.abs((np.random.randn(1,1)).round(1))
    Valid_Execution_Speed+=floated
    return params,Valid_Execution_Speed
def ReinforcementLearning():
    from sklearn.linear_model import LinearRegression
    AI=LinearRegression()
    return AI
def environment():
    params,_=initialization()
    import pandas as pd

    b=pd.DataFrame(params['init_state'],columns=['init_sta
    te'])
    c=pd.DataFrame(params['final_state'],columns=['final_s
    tate'])
    d=pd.DataFrame(params['state_transition'],columns=['st
    ate_transition'])
    e=pd.DataFrame(params['Execution_Start_Time'],columns=
    ['Execution_Start_Time'])
    f=pd.DataFrame(params['Execution_End_Time'],columns=['
    Execution_End_Time'])
    g=pd.DataFrame(params['Calculate_TimeInterval'],column
    s=['Calculate_TimeInterval'])
    data=pd.concat([b,c,d,e,f,g],axis=1)
    columns_train=['init_state','final_state','state_trans
    ition','Execution_Start_Time','Execution_End_Time']
```

```
    column_target=['Calculate_TimeInterval']
    X=data[columns_train]
    Y=data[column_target]
    return X,Y,pd

def agent():
    X,Y,pd=environment()
    RL=ReinforcementLearning()
    RL.fit(X,Y)
    return RL,pd
def intelligence():
    RL,pd=agent()
    import numpy as np

    np.random.seed(1)
    params={}

    params['init_state']=np.abs((np.random.randn(100,1)*2.
    7).round())
    params['final_state']=np.abs((np.random.randn(100,1)*6
    ).round())
    params['state_transition']=np.abs((np.random.randn(100
    ,1)*5).round())
    params['Execution_Start_Time']=np.abs((np.random.randn
    (100,1)*2).round(1))
    params['adder']=np.abs((np.random.randn(100,1)*1.5).ro
    und(1))
```

```
    params['Execution_End_Time']=(params['Execution_Start_
    Time']+params['adder']).round(1)
   #
    a=pd.DataFrame(params['Functional_Unit'],columns=['Fun
    ctional_Unit'])
    b=pd.DataFrame(params['init_state'],columns=['init_sta
    te'])
    c=pd.DataFrame(params['final_state'],columns=['final_s
    tate'])
    d=pd.DataFrame(params['state_transition'],columns=['st
    ate_transition'])
    e=pd.DataFrame(params['Execution_Start_Time'],columns=
    ['Execution_Start_Time'])
    f=pd.DataFrame(params['Execution_End_Time'],columns=['
    Execution_End_Time'])
   # k=pd.DataFrame(Name,columns=['Software'])
    test=pd.concat([b,c,d,e,f],axis=1)

    reward=RL.predict(test)
    #test=pd.concat([test,k],axis=1)
    return reward,test
def upper():
    Name=input('Name of software to be Tested?:')
    TF=eval(input('Total Functional Unit?:'))
    return Name,TF
def subfunctions():
    Name,TF=upper()
```

```
    TotalFunctional_Unit=TF
    SubFunctional_unit=[]
    for i in range(TotalFunctional_Unit):
        SubFunc_unit=float((input("SubFunctional_unit   for
    Functional Unit"+str(i+1)+":")))
        SubFunctional_unit.append(SubFunc_unit)
    return SubFunctional_unit,Name,TF

def Naming():
    Name,TF=upper()
    return Name,TF

def decision():
   # Name,TF=upper()
    #TotalFunctional_Unit=TF
    SubFunction_unit,Name,TF=subfunctions()
    TotalFunctional_Unit=TF
    ExecutionCount=0
    SuccessCount=0
    ErrorCount=0
    Func=[]
    EST=[]
    EET=[]
    Status=[]
    Name=[Name]
    Functional_Unit=0
    #Name=input('Name of software to be Tested?')
```

```
for i in range(TotalFunctional_Unit):
    _,VES=initialization()
    reward,test=intelligence()
   # Name=test['Software'][i]

Execution_Start_Time=test['Execution_Start_Time'][i]
    Execution_End_Time=test['Execution_End_Time'][i]
    reward=reward[i]
    SubFunctional_unit=SubFunction_unit[i]
    #SubFunctional_unit=eval(input("SubFunctional_unit
for Functional Unit"+str(i+1)+":"))
    if ((reward<=VES)and(SubFunctional_unit>3)):
        SuccessCount=SuccessCount+1
        ExecutionCount=ExecutionCount+1
        Functional_Unit=Functional_Unit+1
        Func.append(Functional_Unit)
        EST.append(Execution_Start_Time)
        EET.append(Execution_End_Time)
        Status.append('Execution Successful')
       # print(Name)
       #
print('Execution_Start_Time',Execution_Start_Time)
       #
print('Execution_End_Time',Execution_End_Time)
       # print('Functional_Unit',Functional_Unit)
       # print('Execution Successful')
    elif SubFunctional_unit<=3:
```

```
        Functional_Unit=Functional_Unit+1


    else:
        ErrorCount=ErrorCount+1
        ExecutionCount=ExecutionCount+1
        Functional_Unit=Functional_Unit+1
        Func.append(Functional_Unit)
        EST.append(Execution_Start_Time)
        EET.append(Execution_End_Time)
        Status.append('System Error')
        #print(Name)
       #
print('Execution_Start_Time',Execution_Start_Time)
       #
print('Execution_End_Time',Execution_End_Time)
       # print('Functional_Unit',Functional_Unit)
       # print('System Error')

import pandas as pd
# print('Software:',Name)
# print('Total Functional_Unit:',Functional_Unit)
# print('SuccessCount:',SuccessCount)
# print('ErrorCount:',ErrorCount)
# print('ExecutionCount:',ExecutionCount)
a=pd.DataFrame(Func,columns=['Functional_Unit'])
e=pd.DataFrame(EST,columns=['Execution_Start_Time'])
```

```
    f=pd.DataFrame(EET,columns=['Execution_End_Time'])
    g=pd.DataFrame(Status,columns=['Status'])
    report=pd.concat([a,e,f,g],axis=1)
    Executed_unit=report['Functional_Unit'].count()

    w=pd.DataFrame(["TotalFunctional_Unit","Total_Successf
    ul_Execution","Total_Error_Execution"],columns=['Param
    eters'])

    y=pd.DataFrame([TotalFunctional_Unit,SuccessCount,Erro
    rCount],columns=['Value'])
    dummy=pd.concat([w,y],axis=1)
    software=pd.DataFrame(Name,columns=['software
    tested'])
    #return
    report,dummy,Execution_Start_Time,Execution_End_Time,S
    uccessCount,ErrorCount,ExecutionCount,Functional_Unit,
    Executed_unit,TotalFunctional_Unit
#def Metrics():
    import numpy as np

    import pandas as pd

    #report,dummy,Execution_Start_Time,Execution_End_Time,
    SuccessCount,ErrorCount,ExecutionCount,Functional_Unit
    ,Executed_unit,TotalFunctional_Unit=decision()
```

```
MTTF=np.abs(np.sum((Execution_End_Time-
Execution_Start_Time)/ErrorCount-1))
MTTR=np.round(ErrorCount/ExecutionCount,2)
MTBF=np.round(MTTF+MTTR,2)
Availability=np.round((MTTF/MTBF)*100,0)
MARE=np.round((1/ExecutionCount)*np.sum(((TotalFunctio
nal_Unit-Executed_unit)/TotalFunctional_Unit)*100),0)
RMSE=np.round(np.sqrt((1/ExecutionCount)*np.sum((Total
Functional_Unit-Executed_unit)**2)),2)
FunctionalPoint=np.round(TotalFunctional_Unit*(0.65+0.
01*np.sum(np.random.randint(1,14))),2)
TD=np.round(MTTR,2)
TDD=np.round(TD/250,2)
Portability=np.round((SuccessCount/TotalFunctional_Uni
t)*100,0)
Reliability=np.round(1-(ErrorCount/ExecutionCount),2)
Usability=np.round(SuccessCount/ExecutionCount,2)
Efficiency=np.round(RMSE,2)
Maintainability=np.round(TD,2)
Functionality=np.round(FunctionalPoint,2)

####EVALUATION OF QUALITY
QUALITYeval=((Reliability+Usability+Efficiency+Functio
nality+(Maintainability*60)+(Portability/100))/5)*
((math.log(len(arr)))/(len(arr)-1)**1.25)

X=["Mean Time To Failure(MTTF)","Mean Time To
Repair(MTTR)","Mean Time Between
```

```
    Failure(MTBF)","Availability","SuccessCount","ErrorCou
    nt","ExecutionCount","Mean      Absolute      Relative
    Error","Root                Mean                Square
    Error","FunctionalPoint","TechnicalDebt","TechnicalDeb
    t_Density","Portabiity"]

    Y=[MTTF,MTTR,MTBF,Availability,SuccessCount,ErrorCount
    ,ExecutionCount,MARE,RMSE,FunctionalPoint,TD,TDD,Porta
    bility]
    a=pd.DataFrame(X,columns=['Metric'])
    e=pd.DataFrame(Y,columns=['Values'])

    M=["Reliability","Usability","Efficiency","Functionali
    ty","Maintainability","Portability(%)"]

    N=[Reliability,Usability,Efficiency,Functionality,Main
    tainability,Portability]
    h=pd.DataFrame(M,columns=['Quality_Attribute'])
    g=pd.DataFrame(N,columns=['Values'])
    delta=pd.concat([a,e],axis=1)
    Beta=pd.concat([h,g],axis=1)
    return delta,Beta,report,dummy,software,Name

def Naming():
    Namer=input("DataBase is requesting...Name of software
    to be stored?:")
    Name=str(Namer)
```

```
    return Name
    Name=Naming()

def softwareToSql(dataframe):
    import MySQLdb
    import mysql.connector
    from mysql.connector import Error
    try:
        conn
    =MySQLdb.connect("127.0.0.1","root","Isomeric24","TEST
    DB")
        cur=conn.cursor()
        sql = """CREATE TABLE IF NOT EXISTS Software_{} (
        software CHAR(200))""".format(Name)
        cur.execute(sql)
        main="delete from software_{}".format(Name)
        cur.execute(main)
        conn.commit()
        for(row,rs) in dataframe.iterrows():
            software=rs[0]
            query=("insert  into  Software_{}  values('"+
    software +"')").format(Name)
            cur.execute(query)
        conn.commit()
        cur.close()

    except Error as e:
```

```
        print("Error in Mysql connection=",e)
    finally:
        conn.close()
def reportToSql(dataframe):
    #Name=Naming()
    import MySQLdb
    import mysql.connector
    from mysql.connector import Error
    try:
        conn
    =MySQLdb.connect("127.0.0.1","root","Isomeric24","TEST
    DB")
        cur=conn.cursor()
        sql = """CREATE TABLE IF NOT EXISTS report_{} (
         Functional_Unit  CHAR(100),
         Execution_Start_Time Float,
         Execution_End_Time Float,
         Status CHAR(100))""".format(Name)
        cur.execute(sql)
        main="delete from report_{}".format(Name)
        cur.execute(main)
        conn.commit()
        for(row,rs) in dataframe.iterrows():
            Functional_Unit=str((rs[0]))
            Execution_Start_Time=str(float(rs[1]))
            Execution_End_Time=str(float(rs[2]))
            Status=rs[3]
```

```
        query=("insert   into   report_{}   values('"+
    Functional_Unit  +"','"+  Execution_Start_Time  +"','"+
    Execution_End_Time +"','"+ Status +"')").format(Name)
        cur.execute(query)
        conn.commit()
        cur.close()

    except Error as e:
        print("Error in Mysql connection=",e)
    finally:
        conn.close()
        return

def dummyToSql(dataframe):
   # Name=Naming()
    import MySQLdb
    import mysql.connector
    from mysql.connector import Error
    try:
        conn
    =MySQLdb.connect("127.0.0.1","root","Isomeric24","TEST
    DB")
        cur=conn.cursor()
        sql = """CREATE TABLE IF NOT EXISTS dummy_{} (
         Parameters  CHAR(100),
         Value Float)""".format(Name)
        cur.execute(sql)
```

```
        main="delete from dummy_{}".format(Name)
        cur.execute(main)
        conn.commit()
        for(row,rs) in dataframe.iterrows():
            Parameters=rs[0]
            Value=str(float(rs[1]))
            query=("insert    into    dummy_{}    values('"+
    Parameters +"','"+ Value +"')").format(Name)
            cur.execute(query)
        conn.commit()
        cur.close()

    except Error as e:
        print("Error in Mysql connection=",e)
    finally:
        conn.close()

def deltaToSql(dataframe):
    #Name=Naming()
    import MySQLdb
    import mysql.connector
    from mysql.connector import Error
    try:
        conn
    =MySQLdb.connect("127.0.0.1","root","Isomeric24","TEST
    DB")
        cur=conn.cursor()
```

```
        sql = """CREATE TABLE IF NOT EXISTS delta_{} (
         Metric  CHAR(100),
         Value Float)""".format(Name)
        cur.execute(sql)
        main="delete from delta_{}".format(Name)
        cur.execute(main)
        conn.commit()
        for(row,rs) in dataframe.iterrows():
            Metric=rs[0]
            Value=str(float(rs[1]))
            query=("insert into delta_{} values('"+ Metric
    +"','"+ Value +"')").format(Name)
            cur.execute(query)
        conn.commit()
        cur.close()

    except Error as e:
        print("Error in Mysql connection=",e)
    finally:
        conn.close()

def BetaToSql(dataframe):
    #Name=Naming()
    import MySQLdb
    import mysql.connector
    from mysql.connector import Error
    try:
```

```
        conn
    =MySQLdb.connect("127.0.0.1","root","Isomeric24","TEST
    DB")
        cur=conn.cursor()
        sql = """CREATE TABLE IF NOT EXISTS Beta_{} (
         Quality_Attribute CHAR(100),
         Value Float)""".format(Name)
        cur.execute(sql)
        main="delete from  Beta_{}".format(Name)
        cur.execute(main)
        conn.commit()
        for(row,rs) in dataframe.iterrows():
            Quality_Attribute=rs[0]
            Value=str(float(rs[1]))
            query=("insert           into           Beta_{}
    values('"+Quality_Attribute        +"','"+        Value
    +"')").format(Name)
            cur.execute(query)
        conn.commit()
        cur.close()

    except Error as e:
        print("Error in Mysql connection=",e)
    finally:
        conn.close()
def execute():
```

```
    #report,dummy,Execution_Start_Time,Execution_End_Time,
    SuccessCount,ErrorCount,ExecutionCount,Functional_Unit
    ,Executed_unit,TotalFunctional_Unit=decision()
    #delta,Beta=Metrics()
    delta,Beta,report,dummy,software,_=decision()
    softwareToSql(software)
    reportToSql(report)
    dummyToSql(dummy)
    deltaToSql(delta)
    BetaToSql(Beta)
Name=Naming()
execute()
print("Done.........")
```