International Journal of Computer Applications Technology and Research
Volume 6–Issue 4, 175-182, 2017, ISSN:-2319–8656

Software Quality Measure

Eke B. O.
Department of Computer Science
University of Port Harcourt
Port Harcourt, Nigeria

Musa M. O.
Department of Computer Science
University of Port Harcourt
Port Harcourt, Nigeria

Abstract: Modern gadgets and machines such as medical equipments, mobile phones, cars and even military hardware run on software. The operational efficiency and accuracy of these machines are critical to life and the well being of modern civilization. When the software powering these machines fail it exposes life to danger and can cause the failure of businesses. In this paper, software quality measure is presented with the emphasis on improving standard and controlling damages that may result from badly developed application. The research shows various software quality standards and quality metrics and how they can be applied. The application of the metrics in measuring software quality in the research produced results which shows that the code metrics performance is better than the design metrics performance and points to a new way of improving quality by refactoring application code instead of developing new designs. This is believed to ensure reusability and reduced failure rate when software is developed.

Keywords: Software, quality, reusability, metrics, measure

1. INTRODUCTION

Software quality measures how well software is designed (*quality of design*), and how well the software conforms to that design (*quality of conformance*), although there are several different definitions. It is often described as the 'fitness for use for the purpose' of developing a piece of software. Whereas *quality of conformance* is concerned with implementation, *quality of design* measures how valid the design and requirements are in creating a worthwhile product. But what exactly is software quality? It's not an easy question to answer, since the concept means different things to different people.

Software quality may be defined as the degree of conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software (Ho-Won, et al. 2014). In the definition, it is clear that software requirements are the foundations from which quality is measured. It is then believed that lack of conformance to requirement is lack of quality. Specified standards define a set of development criteria that guide the management of software engineering. Hence, if criteria are not followed during software development, lack of quality will usually result.

A set of implicit requirements often goes unmentioned, for example ease of use, maintainability, usability and other software quality concerns. If software confirms to its explicit (clearly defined and documented) requirement but fails to meet implicit (not clearly defined and documented, but indirectly suggested) requirements, software quality is suspected.

As with any definition, the definition of 'software quality' is also varied and debatable. Some even say that 'quality' cannot be defined and some say that it can be defined but only in a particular context. Some even state confidently that 'quality is lack of bugs'. Whatever the definition, it is true that quality is something we all aspire to have when developing software .

The Institute of Electrical and Electronics Engineers (IEEE) defines software quality as *t*he degree to which a system, component, or process meets specified requirements and the degree to which a system, component, or process meets customer or user needs or expectations.

Similarly, International Software Testing Qualifications Board (ISTQB) defines software quality as the degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations. The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs (Stephen, 2012).

2. SOFTWARE DEVLOPMET LIFE CYCLE

When developing software of high quality, it is crucial to have a good understanding and knowledge of the various phases or stages of Software Development Life Cycle (SDLC). Software Development Life Cycle, or Software Development Process, defines the steps/ stages/ phases in the building of quality software (McConnell, 2015).

There are various kinds of software development models like:

i) Waterfall model
ii) Spiral model
iii) Iterative and incremental development (like 'Unified Process' and 'Rational Unified Process')
iv) Agile development (like 'Extreme Programming' and 'Scrum')

Models are evolving with time and the development life cycle can vary significantly from one model to the other. However, each model comprises of all or some of the core phases/ activities/ tasks involved in software development.

2.1 The Basic Model of SDLC

The basic model of the Software development Life Cycle starts out with the requirement analysis and moves into the design phase, the implementation phase, testing phase, the release phase and cycles back to the requirement phase (Scott, 2005).

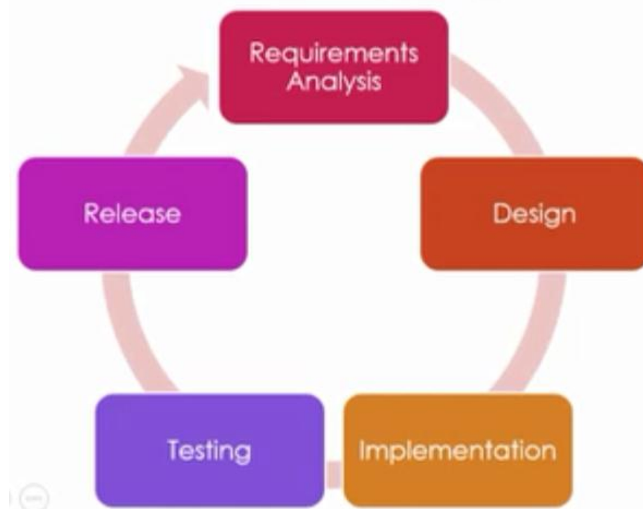

Figure 1: SDLC Basic Model

The phases specified in figure 1 is basic and its arrangement may vary from one methodology to another, however the activities carried out in the phases are similar.

Activities in the life cycle:

Requirement : In requirement activity, developers work directly with customer(s) and identifies the problem to be solved. It focuses on "what" the software is intended to do and not "how". It is important to note that often what a client or customer actually need is often not very clearly expressed and often vary within the development period.

Analysis and Design: This activities focuses on how the programme will achieve the software requirements. The activities start from analysis by making sure that the problem is broken down into smaller pieces called components and then the design is carried out when components are used in synthesizing the system to work together to make the whole programme work.

Implementation : In this phase the code is written according to design specifications. The implementation language may be a barrier to the development of the system. The developers must select a programming language that will be able to handle all of the concerns in requirement captured in the design. This is important due to the fact that certain compiler restrictions or implementation may not allow easy development of certain components according to specification. The selection of programming language of development is therefore an important consideration when quality of application is considered at the implementation stage of software development life cycle. Some time if a reusable software component is available it is preferable to reuse it if it had been previously tested to be working efficiently.

Testing: During the testing activity the code or the design is verified by using different ways in checking if the design or the code met certain the design specification or code functional specification. It is a strong view held by software engineers that if proper testing is carried out at the design stage of a software development life cycle then the coding testing will only be a confirmatory test that the system is working properly as expected. This view have been researched upon to even check whether it is better to carry-out design testing before code testing and which of the two is capable of revealing development error. A similar verification was carried out by Jiang in one of the researches (Jiang et. al., 2007).

Release: When software is released certain concern and requirement may be omitted by the developer or the customer. When the requirement is omitted by the customer it may be released in the next version of the software and it is often not held as a quality issues rather it is an upgrade issue. However, when the requirement omission is from the developer it is a serious quality concern issue. It is at the release phase that the software is closely examined by the staff of the customer(s) and validates that the programme meets the customer's expectations.

There may still be many other activities/ tasks which have not been specifically mentioned above depending on the software design methodology. But it is essential that the key activities within a software development life cycle be understood even if it is at a review level.

2.2 Who Cares About Software Quality?

With software or anything else, assessing quality means measuring value. Something of higher quality has more value than something that's of lower quality (IOS, 2010). Yet measuring value requires answering another question: value to whom? In thinking about software quality, it's useful to focus on three groups of people who care about its value, as Figure 2 shows.

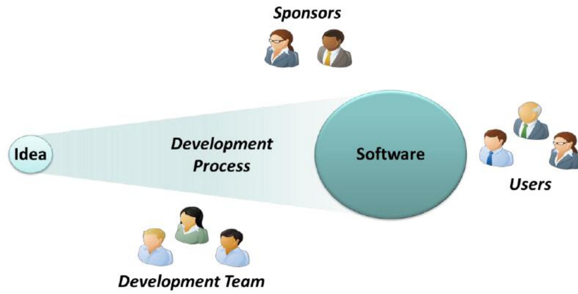

Figure 2: Those who care about software quality

As the figure illustrates, a development process converts an idea into usable software. The three groups of people who care about the software's quality during and after this process are:

1. The software's *users*, who apply this software to some problem.
2. The *development team* that creates the software.
3. The *sponsors* of the project, who are the people paying for the software's creation. For software developed by an organization for its own use, for example, these sponsors are commonly business people within that organization.

All three of these groups are stakeholders of software quality. The aspects of quality that each finds most important aren't the same, however. Understanding these differences requires dissecting software quality to really see the detail structure.

3. ANALYSIS OF SOFTWARE QUALITY

Analysis involves the decomposition of the system into its component parts to identify the part that can be combined in forming a new system. Hence it is useful to think about the software quality by dividing it into three aspects: functional quality, structural quality, and process quality. Doing this helps us see the big picture, and it also helps clarify the trade-offs that need to be made among competing goals (Basili, et, al.,1996). . Figure 4 illustrates this idea.

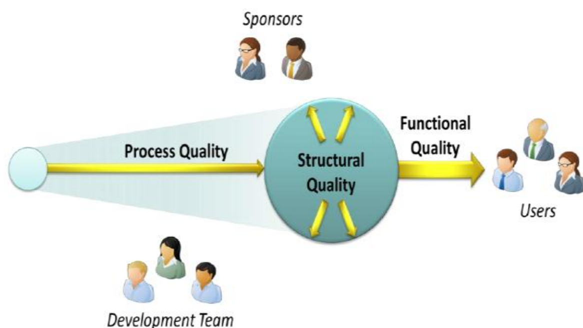

Fig.3: Software quality decomposed into three aspects: functional quality, structural quality, and process quality.

The three aspects of software quality are *functional* quality, *structural* quality, and *process* quality.

3.1 Functional Quality

Functional quality reflects how well the software complies with or conforms to a given design, based on functional requirements or specifications. This attribute also ensures that the software correctly performs the tasks it's intended to do for its users. Among the attributes of functional quality are:

1. ***Meeting the specified requirements.*** Whether they come from the project's sponsors or the software's intended users, meeting requirements is the *sine qua non* of functional quality. In some cases, this might even include compliance with applicable laws and regulations. Requirements commonly change throughout the development process, achieving this goal requires the development team to understand and implement the correct requirements throughout, not just those initially defined for the project.
2. ***Creating software that has few defects.*** Among these are bugs that reduce the software's reliability, compromise its security, or limit its functionality. Achieving zero defects is too much to ask from most projects, but users are rarely happy with software they perceive as buggy.
3. ***Good enough performance.*** Users often perceive slow system as not been well designed or to be outrightly a bad software. The thing that may be causing the low performance might be very simple but that is not actually what the user sees. It is the end product of the software that the user interacts with.
4. ***Ease of learning and ease of use.*** To its users, the software's user interface *is* the application, and so these attributes of functional quality are most commonly provided by an effective interface and a well-thought-out user workflow. The aesthetics of the interface—how beautiful it is—can also be important, especially in consumer applications.

Software testing commonly focuses on functional quality. All the characteristics just listed can be tested, at least to some degree, and so a large part of ensuring functional quality boils down to testing.

3.2 Structural Quality

The second aspect of software quality, structural quality, means that the code itself is well structured. Unlike functional quality, structural quality is hard to test for (although there are tools to help measure it) (Robert, 1992). The attributes of this type of quality include:

1. ***Code testability.*** Checking if the developed code is organized in a way that makes testing easy or whether testing the code will be fell based on the style of code development.

2. ***Code maintainability.*** High level modularity is also checked to make sure that it is easy to add new code or change existing code without introducing bugs in other part of the program.
3. ***Code understandability.*** Is the code readable? Is it more complex than it needs to be? These have a large impact on how quickly new developers can begin working with an existing code base.
4. ***Code efficiency.*** It also check if the program consumes a lot of system resources in execution, and writing efficient code can be critically important in making the application to execute in old and newer machines. Users often do not need to upgrade their hardware or to buy new system just to be able to run a program, when similar app can also run in their machine.
5. ***Code security.*** Does the software allow common attacks such as buffer overruns and SQL injection? Is it insecure in other ways?

3.3 Process Quality

Process quality, is also critically important. The quality of the development process significantly affects the value received by users, development teams, and sponsors, and so all three groups have a stake in improving this aspect of software quality(Antoniol, et, al.,2002)..

The most obvious attributes of process quality include these:

1. ***Meeting delivery dates.*** Was the software delivered on time?
2. ***Meeting budgets.*** Was the software delivered for the expected amount of money?
3. ***A repeatable development process that reliably delivers quality software.*** If a process has the first two attributes—software delivered on time and on budget—but so stresses the development team that its best members quit, it isn't a quality process. True process quality means being consistent from one project to the next.

4. SOFTWARE QUALITY ASSURANCE

Software Quality Assurance (SQA) is a set of activities for ensuring quality in software engineering processes (that ultimately result in quality in software products). These activities include:

Process definition and implementation, Auditing, and Training

Processes could be:

1. Software Development Methodology
2. Project Management
3. Configuration Management
4. Requirements Development/Management
5. Estimation
6. Software Design
7. Testing, etc.

Once the processes have been defined and implemented, Quality Assurance has the following responsibilities:

1. identify weaknesses in the processes
2. correct weakness to continually improve the process

The quality management system under which the software system is created is normally based on one or more of the following models/standards which are the most popular models:

1. CMMI 2. Six Sigma 3. ISO 9000

There are many other models/standards for quality management but the ones mentioned above are the most popular. Software Quality Assurance encompasses the entire software development life cycle and the goal is to ensure that the development and/or maintenance processes are continuously improved to produce products that meet specifications/requirements. The process of Software Quality Control (SQC) is also governed by Software Quality Assurance (SQA). SQA is generally shortened to just QA.

4.1 Software Quality Control

Software Quality Control (SQC) is a set of activities carried out to ensure quality in software products (Antoniol, et, al.,2001).

It includes the following activities:

i) *Reviews:* The review of the activities carried out must be done at all stages of the life cycle based on the methodology selected for the system development. In the sample model we are using in this paper it may include:

1. Requirement Review: Review carried out when the initial requirements have been done, to check if all the requirements needed in the system are captured.
2. Design Review: When the design of the system is completed, the review re-examine the design to see if there are certain omissions that needed to be corrected.
3. Code Review: This involve the checking of the coding pattern to see if it satisfies the principles required for quality program.
4. Deployment Plan Review : The review is carried out to make sure that there are no omissions in the plans for the deployment of the system.
5. Test Cases and Test Plan Review involve the checking of the test conditions required to execute the system.

ii). *Testing:* Testing varies from one methodology to another but one issue is common to them all, which is that testing need to be done. Some methodology reserve a specific time for testing phase while other encourage progressive testing throughout the life cycle. Whichever process that is used some of the testing carried out include:

1. Unit Testing: This involve the testing of single modules or program units and to make sure that it is working according to the expected goal.
2. Integration Testing: Once the units are working according to the expectation they can be brought

together and tested to make sure they are working well as a whole unit.

3. System Testing: once the entire system is ready for deployment it can still be tested with varying example data to make sure that various input data will work up to the expectation of the system.
4. Acceptance Testing: In this stage the customers can use the real life data set to test the system before it is finally deployed for usage.

Software Quality Control is limited to the Review/Testing phases of the Software Development Life Cycle and the goal is to ensure that the products meet specifications/requirements. The process of Software Quality Control (SQC) is governed by Software Quality Assurance (SQA). While SQA is oriented towards prevention, SQC is oriented towards detection. Some people assume that QC means just Testing and fail to consider Reviews; this should be discouraged (Schröter, et al., 2006).

Differences between Software Quality Assurance (SQA) and Software Quality Control (SQC)

Criteria	**Software Quality Assurance (SQA)**	**Software Quality Control (SQC)**
Definition	SQA is a set of activities for ensuring quality in software engineering processes (that ultimately result in quality in software products). The activities establish and evaluate the processes that produce products.	SQC is a set of activities for ensuring quality in software products. The activities focus on identifying defects in the actual products produced.
Focus	Process focused	Product focused
Orientation	Prevention oriented	Detection oriented
Breadth	Organization wide	Product/project specific
Scope	Relates to all products that will ever be created by a process	Relates to specific product
Activities	i) Process Definition and Implementation ii) Audits iii) Training	i) Reviews ii) Testing

5. SOFTWARE TESTING

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Statistics had been used over the year in test (Siegel, 1956) and it is still been used in certain parameter test even in software metrics. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects).

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest (Diomidis, 2006).. In general, these properties indicate the extent to which the component or system under test:

1. meets the requirements that guided its design and development,
2. responds correctly to all kinds of inputs,

iii) performs its functions within an acceptable time,
iv) is sufficiently usable,
v) can be installed and run in its intended environments, and
vi) achieves the general result its stakeholders desire.

As the number of possible tests for even simple software components is practically infinite, all software testing uses some strategy to select tests that are feasible for the available time and resources. As a result, software testing typically (but not exclusively) attempts to execute a program or application with the intent of finding software bugs (errors or other defects). The job of testing is an iterative process as when one bug is fixed, it can illuminate other, deeper bugs, or can even create new ones.

5.1 Software Testing Levels

There are four levels of software testing: *Unit >>Integration >>System >>Acceptance*.

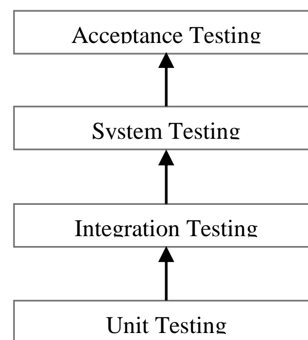

Fig. 4: A Software Testing level

1. Unit Testing is a level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.
2. Integration Testing is a level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
3. System Testing is a level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. Acceptance Testing is a level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Note: Some tend to include Regression Testing as a separate level of software testing but that is a misconception. Regression Testing is, in fact, just a type of testing that can be performed at any of the four main levels.

5.2 Techniques of Software Testing

Below are some methods / techniques of software testing:

1. Black Box Testing is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional. Test design techniques include: *Equivalence partitioning, Boundary Value Analysis, Cause Effect Graphing.*
2. White Box Testing is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. Test design techniques include: *Control flow testing, Data flow testing, Branch testing, Path testing.*
3. Gray Box Testing is a software testing method which is a combination of Black Box Testing method and White box Testing method.
4. Agile Testing is a method of software testing that follows the principles of agile software development.
5. Ad Hoc Testing is a method of software testing without any planning and documentation.

6. SOFTWARE ENGINEERING STANDARDS TEST MEASURE

According to the IEEE Comp. Soc. Software Engineering Standards Committee, a standard can be: An object or measure of comparison that defines or represents the magnitude of a unit. It can also be a characterization that establishes allowable tolerances or constraints for categories of items, or a degree or level of required excellence or attainment.

6.1 Software Standards Legal Implications

Comparatively few software products are forced by law to comply with specific standards, and most have comprehensive non-warranty disclaimers. However, for particularly sensitive applications (e.g. safety critical) software will have to meet certain standards before purchase.

1. Adherence to standards is a strong defence against negligence claims (admissible in court in most US states).

2. There are instances of faults in products being traced back to faults in standards, so

3. Standards writers must themselves be vigilant against malpractice suits.

When standards are released, it is also important to subject the so call standard to QA testing to make sure that serious fault will not arise by adhering to those standards.

6.2 Quality Assurance Standards

Differing views of quality standards: taking a systems view (that good management systems yield high quality); and taking an analytical view (that good measurement frameworks yield high quality). Examples:

1. Quality management: ISO 9000-3 Quality Management and Quality Assurance Standards - Part 3: Guidelines for the application of 9001 to the development, supply, installation and maintenance of computer software

2. Quality measurement: IEEE Std 1061-1992 Standard for Software Quality Metrics Methodology

6.2.1 Product Standards

These focuses on the products of software engineering, rather than on the processes used to obtain them. Perhaps surprisingly, product standards seem difficult to obtain. Examples:

1. Product evaluation: ISO/IEC 14598 Software product evaluation

2. Packaging: ISO/IEC 12119:1994 Software Packages - Quality Requirements and Testing

6.2.2 Process Standards

A popular focus of standardization, partly because product standardization is elusive and partly because much has been gained by refining process. Much of software engineering is in fact the study of process. Examples:

1. Life cycle: ISO/IEC 12207:1995 Information Technology - Software Life Cycle Processes
2. Acquisition: ISO/IEC 15026 System and software Integrity Levels
3. Maintenance: IEEE Std 1219-1992 Standard for Software Maintenance
4. Productivity: IEE Std 1045-1992 Standard for Software Productivity Metrics.

7 EXPERIMENTAL USE CASE

In the experimental system a project program was used to examine the concepts provided in this paper to empirically find out what will be the result using a given set of data and a varying metrics.

Condition1 : i) Model development using design metrics
ii) Model development using design metrics only

Condition 2: i) Testing the models with data set utilizing code metrics
ii) Testing models with data set utilizing design metrics

Data set: The data set was randomly generated so that various type of data will be covered and the data that may be considered none applicable will also be tested. When the data is not within area that the system should handle the system need to graciously handle such challenge without an outright crash.

Metrics Used: The metrics used include selected design metrics and selected code metrics.

Design Metrics: The design metrics used in the experimentation (Subramanyam et,al., 2003) include: The design complexity of a module, Design_Density, Essential_Complexity Module, Essential_Density and Maintenance_Severity. All the design metrics were calculated as a factor of cyclomatic complexity of a module $(e - n + 2)$ where n could be Number of calls to other functions in a module and e effort matrics of the module.

Code Metrics: The design metrics used in the experimentation include: The halstead length content of a module $\mu = \mu 1 + \mu 2$,
The halstead length metric of a module $N = N1 + N2$,
The halstead level metric of a module $L = (2*\mu_2) / (\mu_2*N_2)$
The halstead difficulty metric of a module $D = 1/L$
The halstead volume metric of a module $V = N * \log 2(\mu 1 + \mu 2)$
The halstead effort metric of a module $E = V/L$
The halstead programming time metric of a module $T = E/18$
The halstead error estimate metric of a module $B = E^{2/3}/1000$

Method: In the experimental use case six specific instruments or programs are selected for quality examination and the design metrics as well as the code metrics was used to test the outcome using the various data set. A fault tool checker was also deployed to compare the result from the metrics to the result from the tool checker is statistically obtained via the internal system of the tool and the percentage fault was also extracted.

7.1 Results

The result of the quality test is clearly displayed on the table. The result show a note of the specific intrument of program module used in the test. It also show the performance of the metrics using their fault levels. The instruments are different and that variation is also reflected in the data sets used in testing the system table 1 clearly illustrate all these values.

Table 1: Result of percentage fault from the tests

Data set	Test No	% Fault		Note
		Design metrics	Code metrics	Specific Instrument
DT1	0001	2.1	0.7	Simple number computation
DT2	0002	5.4	1.2	Input and output processing
DT3	0003	2.5	0.9	A database system
RD1	0004	14.2	7.3	A combustion experiment
RD2	0005	7.8	3.2	Multimedia system
RD3	0006	16.3	2.8	A Recursive procedure

In figure 5 it is clear from the graph the percentage fault of each of the metrics used. The design metrics performance of the code metrics appears to be better than the performance of the *design* metric groups. In the 6 data sets, the instruments where clearly varied from simple input-output processing which is a very simple program that can be easily tested for performance to recursive procedures which if not controlled could result to a forever executing system that can exhaust the processing resources and memory of the system if continuous output is generated.

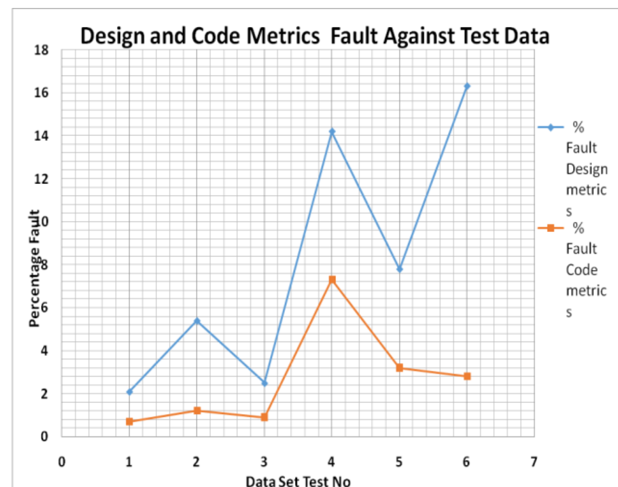

Fig. 5: A plot of the testing result

The fault was lowest on simple number computation which is understandable but on the contrary instead of recursive procedure showing the highest fault level on the code metrics it was the combustion engine classical program that was not written in a highly modular format that showed the highest fault level for the code metrics. It is clear therefore that it is not only the metrics that are contributing to the fault that the process type used in the development of the system that also contribute a great part to the system efficiency.

8.CONCLUSION

Organizations that develop low-quality software, whether for internal use or for sale, are always looking backward, spending time and money on fixing defects in "finished" products. In contrast, an organization that builds in product quality from the beginning can be forward-looking and innovative; it can spend its resources on pursuing new innovations instead of spending the time on maintenance.

In the use case test it is clear that the instrument (type of problem solved by a program) alone does not determine the performance of the system. The process or methodologies used clearly contribute much in the performance; a poor process can result to bad program both at design and implementation. The benefits of including quality-oriented activities in all phases of a software development lifecycle are both broad and deep. These measures not only facilitate innovation and lower costs by increasing predictability, reducing risk, and eliminating rework, but they can also help to differentiate an quality product from its competitors. Most important, continuously ensuring quality will always cost less than ignoring quality considerations.

9.RECOMMENDATION

In most of the instruments, it is clear that those that have very bad fault at design also had corresponding higher fault at implementation. The paper did not correlate the two but from the plot of the result the relationship of the two plot is obvious. It is therefore recommended the a combined effort at improving both design fault and coding fault can be a target that can be easily realizable if good process and programming practice is imbibed. Further research is also recommended to find out correlation between the metrics to see the effect or level of fault relationship. This will enable a valuable discuss on the regression test of the design fault with the coding fault. This work is recommended as launch pad to such research so that the quality issues raised and discussed in this work will be used in handling such cases.

ACKNOWLEDGEMENT

Oyol Computer Consult Inc Port Harcourt, Nigeria is acknowledged for providing the facility and tool used in carrying out the experimental tests. We also thank them for offering some of the instruments used.

REFERENCES

Ho-Won J., Seung-Gweon K., and Chang-Sin C. (2014). Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software*, 21(5):10–13, September/October 2014.

Stephen H. K (2012). *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston, MA, second edition.

McConnell, S. (2015), *Code Complete* (Fifth ed.), Microsoft Press Pressman,

Scott M. (2005), *Software Engineering: A Practitioner's Approach* (Sixth, International ed.), McGraw-Hill Education

Jiang Y., Cukic, B. and Menzies T. (2007) Fault prediction using early lifecycle data. pages 237–246. Software Reliability. ISSRE '07. The 18th IEEE International Symposium on, Nov. 2007.

International Organization for Standardization.(2010) *Software Engineering—Product Quality—Part 1: Quality Model*. ISO, Geneva, Switzerland, 2010. ISO/IEC 9126-1:2010(E).

Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo.(2002) Recovering tracebility links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983.

Antoniol, G., Casazza, G., Penta, M. and Fiutem, R. (2001) Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196.

Basili, V. R., Briand L. C. and Melo W. L. (1996). A validation of object-oriented design metrics as quality indicators, 1996.

Breiman. L (2001) Random forests. *Machine Learning*, 45:5– 32, 2001.

Diomidis S.(2006). *Code Quality: The Open Source Perspective*. Addison Wesley, Boston, MA, 2006.

Robert L. Glass.(1992) *Building Quality Software*. Prentice Hall, Upper Saddle River, NJ, 1992.

Roland Petrasch, (1999) "The Definition of, Software Quality': A Practical Approach", ISSRE, 1999

Schröter, A, Zimmermann, T and Zeller A. (2006) Predicting component failures at design time. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering*, pages 18–27, New York, NY, USA, 2006. ACM Press.

Siegel. S. (1956) *Nonparametric Satistics*. New York: McGraw- Hill Book Company, Inc., 1956.

Subramanyam R. and M. S. Krishnan. M S. (2003) Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310,