Software Evolution: Improving the Quality and Maintainability of Software

Edward E. Ogheneovo
Department of Computer Science
University of Port Harcourt
Port Harcourt, Nigeria
E-mail: edward_ogheneovo@yahoo.com

ABSTRACT

Software evolution is the process by which programs change shape, adapt to the marketplace and inherit characteristics from preexisting programs. As large-scale programs such as Windows and Solaris expand well into the range of 30 to 50 million lines of code, project managers have devoted much of their time to working on legacy codes by adding new functionalities and making the existing codes more structured and providing better documentations for users. Thus the resultant evolution of software appears to be driven and controlled by human decision, managerial ethic, and programmer's judgment. Software evolution takes place when the initial development was successful. The goal is to adapt the application to the ever-changing user requirements and operating environment. The evolution stage also corrects the faults in the application and responds to both developer and user learning. This paper discuss software evolution, evolutionary process, how evolution permits building better concepts in software development process. Finally, we argue that software evolution helps to improve software quality and provides easy and better maintainability for future software maintainers.

Keywords: Software evolution, software maintenance, software, evolutionary process, software maintainers

iSTEAMS Cross-Border Conference Proceedings Paper Citation Format
Edward E. Ogheneovo (2018) Software Evolution: Improving the Quality and Maintainability of Software. Proceedings of the 13th iSTEAMS Multidisciplinary Conference, University of Ghana, Legon, Accra, Ghana. Vol. 2, Pp 185-194

1. INTRODUCTION

Software systems are ubiquitous these days. They are used in virtually all areas of human endeavour may it be electrical appliances, manufacturing industries, automobiles, schools and universities, military and war equipments, health care, finance, government, entertainment, etc. A typical software system is modified and extended a number of times during its lifetime to keep it operational [1]. In fact, majority of software engineers these days do not develop new software but are involved in extending or maintaining the existing ones usually referred to as legacy code. Lehman's first law of evolution caps it all when it states that a software system that is used in a real-world environment must change or it becomes progressively less useful in that environment [2] [3] [4]. Software must therefore be developed in such a way that it can easily be amenable and evolve after its initial development.

Software systems are continually changing and adapting to meet the needs of their users and surrounding environments in terms of new functionalities, new architecture, new technological support, etc. [5] Therefore, a good understanding of the evolution process is essential. Also important is the good understanding of how software evolution and how it helps to improve the qualities and maintainability of software. This permits building better concepts, methods and tools to assist developers as they maintain and enhance these systems. Furthermore, it paves way for the investigation of techniques and approaches to monitor, plan and predict a successful evolutionary paths for long lived software projects. The major goals of software engineering are to enable developers to construct systems that reliably meet user's demand, adapt to changing environment, cost effective, and easily maintainable. The term software evolution relates to the activity and phenomenon of software change [6]. It includes two aspects that reflect, respectively, the complementary concerns of the 'how' and the 'what/why' of software evolution. Interest in the former is concerned with methods, tools and techniques to change functional, performance and other characteristics of the software in a controlled, reliable, fast and cost effective manner [7] [8].

Evolution is a critical issue in the life cycle of all software systems. The process of software evolution is driven by requests for changes and includes change impact analysis, release planning, and change implementation. The resultant evolution appears to be driven and controlled by human decision, managerial ethics, and programmer judgement [9]. Software evolution is the process of conducting continuous software reengineering. Reengineering implies a single change cycle, but evolution can go on forever. In other words, to a large extent, software evolution is a repeated software reengineering process.

To justify the need for maintenance and thus maintainability, it is important to understand how software evolution works. The theory of software evolution shows that even if the original implementation is correct and fully covers the requirements, the software has to be updated regularly for it to stay satisfactory and be able to meet new challenges and advancement in technological innovations. Otherwise, the software becomes progressively less useful in terms of its functionalities.

Software maintainability is an important quality attribute of software systems. It is defined by IEEE as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [10]. Associated with software maintainability is software maintenance (Riaz et al., 2009), which has long been known to consume the majority of the cost of a software development life cycle [12] [13].
In this paper, we study the evolution of software with the aim to discussing how software evolves, and how software evolution is used to improve the quality and maintainability of software as they evolve. The rest of the paper is as follows: In section 2, we discussed the background of software evolution, how software evolve, types of software, software systems and the laws of evolution. In section 3, we discussed software maintenance, characteristics of software maintenance and evolution, types of software maintenance, software maintainability. Section 4 discusses software evolutionary process. Finally, section 5 draws the conclusion.

2. Background of Software Evolution

As large-scale programs such as Windows and Solaris expand well into the range of 30 to 50 million lines of code, project managers have devoted much of their time to working on legacy code by adding new functionalities and making the existing code more structured and providing better documentations for users to be able to use such software [14] [15]. Lehman and Ramil [15] define software evolution as all programming activities that are intended to generate a new software version from an earlier operational version. Chapin et al. [16] defines software evolution as the application of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version together with the associated quality assurance activities and processes, and with the management of these activities and processes.
Software evolution management is a crucial sub-domain of information system engineering tackling the problems associated with post development changes.. Software evolution may occur at random intervals of time, generating new cycles of change in design, implementation and retest.

Software evolution is a continuing process that encompasses not only development but all activities, enhancement, adaptation, or fixing of bugs that occurs after the first operational release. The term software evolution is often preferred to maintenance in post release activity since software does not deteriorate through use but evolved through maintenance in order to satisfy users with respect to the changing operational domain or purpose [17]. The term maintenance is better used for hardware artefacts (e.g. car or generator engine). The principal aim of maintenance is to compensate for wear, tear, and material deterioration so as to restore the condition or performance of an earlier or even original state. But software as said earlier does not wear or deteriorate. Thus evolution implies that the software system must continually be changed to be able to adapt to its new environment.

However, the term maintenance is now used in software engineering the same way as evolution. Therefore, software evolution includes all post first release activities after it has been delivered to the customer. It is a fact of life for E-type software (i.e., software used to solve problems in a real world domain.) must be reliable and perform satisfactorily. The ultimate goal of software evolution is for customer's satisfaction. The process of software evolution

usually spans many year or decades in some situations. As a result, it is generally agreed that the effort needed to maintain software often exceed that needed to develop the same software.

This is why many researchers in the field of software maintenance often estimate the maintenance costs of software to be well above 60% of the total development process. Evolution implies that the software system must continually be changed. In addition to its size, software change rate and change activity must be measured and reflected in cost models. Thus software must be well designed and developed to ensure that the maintenance process is easily carried out as the software evolves.

2.1Laws of Software Evolution

The term software evolution relates to the activity and phenomenon of software. It includes two aspects that reflect, respectively, the complementary concerns of the 'how' and the 'what/why'. of software evolution. Interest in the former is concerned with methods, tools, and techniques to change functionality, performance and other characteristics of the software in a controlled, reliable, fast and cost effective manner. This is the more widespread view and is exemplified in.

Interest in the what/why, on the other hand, focuses on understanding the software evolution phenomenon, its underlying causes and drivers, common patterns of evolutionary behavior, and the characteristics of that behavior. This line of investigation, the focus of the FEAST (Feedback, Evolution And Software Technology) was studied at the Department of Computing at Imperial College. Both views, the 'how' and the 'what/why', must be pursued if mastery of the software evolution phenomenon is to be achieved in a world of increasing dependent on computers and software. The following are examples of the type of questions whose answers are pursued under the latter view:

- Why does software evolve?
- Why is it inevitable?
- What are key attributes of the evolution process?
- What is their impact on the software process and its products?
- What are the practical implications of the above on the planning, control and management of software system evolution?

These questions are answered in Lehman's Laws. Lehman and Belady [2] Lehman [7] studied the growth and evolution of some large software projects and they came out with a number of laws which is today referred to as Lehman's laws. These laws are based on the E-type systems, that is, software systems that solve a problem or implement a computer application in the real world. The laws are described in the table 1.

Table 1: Current statement of Lehman's Laws of Evolution

No	Law	Description
I 1974	**Continuing Change**	E-Type systems must be continually adapted else they become progressively less satisfactory in use.
II 1974	**Increasing Complexity**	As an E-Type System is evolved its complexity increases unless work is done to maintain it.
III 1974	**Self Regulation**	Global E-type system evolution processes are self-regulating.
IV 1978	**Conservation of Organizational Stability**	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime.
V 1978	**Conservation of Familiarity**	In general, the incremental growth and long term growth rate of E-type systems tend to decline. That is, as an E-type system evolves all those associated with it: developers, sales personnel, users, etc., must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1991	**Continual Growth**	The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime
VII 1996	**Declining Quality**	The quality of E-type systems will appear to be declining unless rigorously adapted to changes in the operational environment.
VIII 1996	**Feedback System (Recognized 1971, Formulated 1996)**	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

2.2 Software Quality

Quality is an increasingly critical factor as customers become more sophisticated, technology becomes more complex, and the software business becomes more intensely competitive [18]. Defining what constitutes good quality software is not an easy task. This is simply because no single comprehensive and complete standard definition of its lexicon exists in literature. Again, quality is a perceptual, conditional and somewhat subjective attribute and may be understood or defined in different ways. What might be of good quality to one person may be of low quality to another. There are two common aspects of quality: the first has to do with the consideration of the quality of a thing as an objective reality independent of the existence of man. The other has to do with what we think, feel or sense as a result of the objective reality. In other words, there is subjective side of quality. Thus the word quality has multiple meanings, two of these meanings dominate the use of the word: (1) quality as consists of those product features which meet the need of customers and thereby provide product satisfaction (2) quality consists of freedom from deficiencies. This is the reason he defined quality as "fitness for use". However, quality characteristics are analyzed by the developers and their input data are: application structure, its objectives and its results and are materialized through values directly correlated with the satisfaction that users of a product encounters

The difficulty in defining quality is to translate future needs of the user into measurable characteristics, so that a product can be designed and turned out to give satisfaction at a price that the user will pay. This is not easy, and as soon as one feels fairly successful in the endeavour, he/she finds that the needs of the consumer have changed, competitors have moved in, etc., [19]. Therefore, quality is a customer determination, not an engineer's determination, not a marketing determination, nor a general management determination. It is based on the customer's actual experience with the product or service, measured against his/her requirements—stated or unstated, conscious or unconscious, technically operational or entirely subjective—and always representing a moving target in a competitive market. As noted by William A. Foster [20], quality is never an accident. It is always the result of high intention, sincere effort, intelligent direction and skillful execution; it represents the wise choice of many alternatives.

Therefore, we define software quality as the totality of features and characteristics of a software product that satisfy a given set of requirements. Good quality software is generally taken to mean that a software product provides value (satisfaction) to its users, makes a profit, generates few serious complaints, and contribute is some way to the goals of humanity or at least does no harm to the user and the environment to which it is applied [21]. As noted by Ivan and Boja [22], software quality is seen as being composed of a set of components that collect, process, stock and distribute information that is further used as a support in taking decisions and control within an organization. Some of the factors of software quality include reliability, reusability, maintainability, and portability. These quality factors are then broken down into lower level quality criteria which serve as attributes of software [23]. Coenen [24] defines quality as the totality of characteristics of an entity that bears on its ability to satisfy stated and implied needs. It is existence of characteristics of a product which can be assigned to requirements.

In software engineering, software quality refers to two distinct but related concepts: software functional quality and software structural quality.

- **Software functional quality:** This refers to how a software product complies with or conforms to a given design, based on functional requirements or specification. Functional quality is often enforced and measured through software testing.
- **Software structural quality:** This refers to how a software product meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly. Structural quality is evaluated through the analysis of the software inner structure, its source code, at the unit level, the technology level and the system level, which is in effect, shows how its architecture adheres to sound principles of software architecture.

ISO/IEC-9126 standard (ISO/IEC 9126-1) [25] provided six characteristics of software quality as:

- **Maintainability:** This is the effort required to make certain changes
- **Reliability:** This is the capacity to maintain a level of performance
- **Efficiency:** This is the relationship between the level of performance and the volume of resources used
- **Usability:** This is the effort required to use the software product
- **Portability:** This is the ability to be transferred from one work environment to another
- **Functionality:** The satisfaction of requirements set by users

2.3 Software Maintenance and Software Maintainability

Software maintenance is defined as the modification of a software product after delivery to correct faults, to improve performance or other attributes or to adapt the product to changed environment. It is the process of software undergoing modification to code and associated documentation due to a problem or the need for improvement. According to Software Engineering Body of Knowledge (SWEBOK), maintenance is the totality of activities required to provide cost effective support to a software system.

Software maintainability is a key quality attribute of software systems. It is defined by IEEE as the ease with which software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [26]. Associated with software maintainability is software maintenance, which has long been known to consume the large percentage of software life cycle development costs [27]. Software maintainability can significantly impact software costs, therefore, it is imperative to understand software maintainability so as to control and improve it. It is the degree to which it can be understood, corrected, adapted and/or enhanced [28]. Software maintenance accounts for more effort than any other software engineering activity. Maintainability is a key goal that guides software engineering processes. According to the Factor-Criteria-Metric Model, the maintainability factor is comprised of consistency, simplicity, conciseness, self-descriptiveness, and modularity. According to the IEEE Standard for quality metrics, maintainability can be decomposed to correctability, testability, and expandability. Therefore, maintainability consists of analyzability, changeability, stability and testability.

Again we define maintainability as the quality factor which includes all features of the software to make it easier to maintain or which makes the maintenance stage more productive. Thus software maintainability is the degree to which it can be understood, corrected, adapted and/or enhanced. Granja-Alvarez and Barranco-Garcia [29] notes that maintainability is the quality factor including all those software characteristics designed to make the product easier to maintain towards the end of achieving greater efficiency and productivity in the maintenance stage. As noted by Sommerville [30], as new systems are used; new requirements emerge. Therefore, it is important to maintain the usefulness of a system by changing it to accommodate these new requirements.

Maintainable software is software that can be adapted economically to cope with new requirements and where there is a low probability that making changes will introduce new errors into the system.

Maintainability is usually measured as the change in effort, that is, the number of lines changed per class [31]. The maintainability of software systems is a crucial point in the software lifecycle. As society becomes more and more dependent on software and demands that new, more capable software be provided on short cycles, the need for maintainable, reliable software continues to increase. The maintainability of software can best be approached at the level of design and coding. Maintainability involves analyzability, how easy it is to locate the elements or faults to improve on or fix; changeability, how much work is needed to implement a change; stability, how few things break after making changes; and testability, the ability to validate software modifications or changes. In other words, maintainability involves the ability to identify and fix fault(s) in the software system. It is the ease with which a product can be maintained in order to correct defects, meet new requirements, make future maintenance easier, or cope with a changed environment. It is an important characteristic of a software system that is intended to help reduce a system's tendency, and to indicate when it becomes cheaper and less risky to rewrite the code instead of changing it [32].

Therefore, in software engineering, maintainability is required for correction of defects, meeting new requirements, to cope with a changing environment, and modifying a software product with ease. As more programs are developed, the amount of effort and resources expended on software maintenance is growing. The maintainability of software is affected by many factors, such as availability of qualified software staff, the ease of system handling, the use of standardized programming languages, etc. However, careless in design, implementation and testing has an obvious negative impact on the ability to maintain the resultant software.

2.4 Maintainability Attributes

There are four main attributes of maintainability [33]. These are:

- **Analyzability:** This is the capability of software to be diagnosed for deficiencies or causes of failures in the software or for identification of parts requiring modification.
- **Changeability:** The capability of software to enable a specified modification to be implemented.
- **Stability:** This is the ability of software to minimize unexpected effects from software modifications.
- **Testability:** This is the ability of software to validate modified software.

2.5 Measuring Software Maintainability

As stated earlier, software maintainability is an important software quality attribute. Since maintainability is a measure of the ease to modify code; higher maintainability means less time to make a change. Maintainable code is code that has high cohesion and low coupling. That is, code that has high cohesion and low coupling is more likely to be maintainable. Coupling is a measure of how related code is. Low coupling means that changing how A does something should not affect B that uses A. Cohesion is a measure of how related, readable and understandable code is. Generally, lower coupling means high cohesion, and vice versa.

Ensuring maintainability is an important aspect of the software development cycle. Maintainable software will be easier to understand and change correctly. There is no agreement on how to measure maintainability. Welker and Oman [34] suggest measuring software maintainability by using a Maintainability Index (MI). That is to say, the most widely used software metric to quantify maintainability is known as Maintainability Index (MI) which is decided by cyclomatic complexity, lines of code (LOC), and lines of comments. Ramil and Lehman proposed the linear model with effort and proportion of modules changed as the variables while Pressman proposed an effort-based metric, mean-time-to-change (MTTC), to predict maintainability.

Polo et al. [35] proposed using number of modification requests, mean effort per modification request, and type of correction measure maintainability. All these metrics use different parameters to measure software and two or more could be combined in order to achieve a better metric value.

3. SOFTWARE EVOLUTIONARY PROCESS

Software evolution processes differ from one organization to the other depending on the type of software being maintained, the development processes used in an organization, and the people involved in the evolution process. Sometimes, evolution could be formal or informal depending on the organization. If the organization is formal, the evolution process must be done formally by ensuring that there are rules and regulation guiding such process. Such rules and regulations are usually documented based on the software product at each stage of the developmental process. In informal organizations, the process of software evolution involves informal conversation between the software user and the developer.

An evolutionary development process minimizes integration and test risk, and facilities availability of system functionality when it is required. Software evolutionary process involves the fundamental activities of change analysis, release planning, system requirement and releasing of a software system to users or customers. Before evolution process takes place, the user or customer must requests the developer for the change. Once this is agreed upon, the impact analysis of the change is carried out. The impact analysis involves determination of the cost of the change, the effect of the change on the system, and any other factor that should be considered before the change requirement begins. The change process then commence and during this process, there is release plan where the defects are repaired or removed, the software is enhanced and adapted to suite its environment.

These changes are then implemented and further tests are carried out to ensure that the software meets specification needed in the change. During the evolution process, user requirement are analyzed in detail. However, new requirements may emerge during the analysis stage. When this happens, the initial proposed changes are then modified and there could be further discussions with the client before these changes are implemented. Figure 1 shows the various steps in the software evolution process.

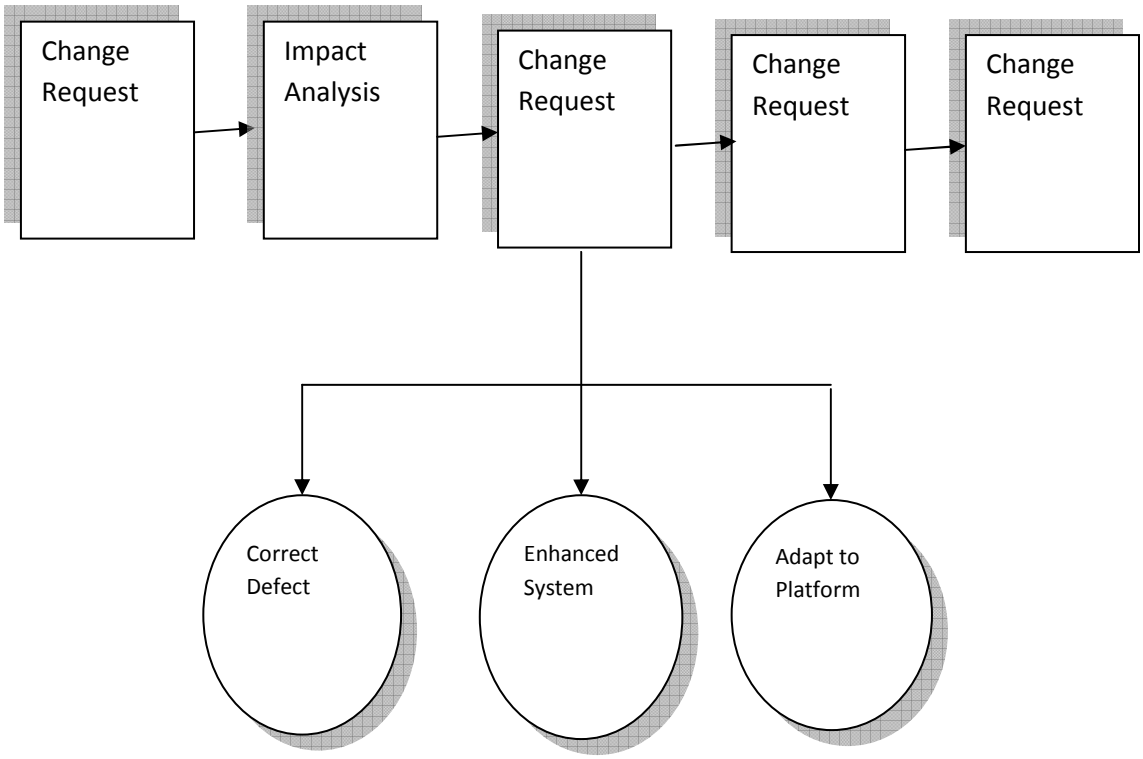

Fig. 1: Software Evolution Process
Source: Sommerville (2007)

During the change implementation requirements, new changes can be identified and then corrected. Thereafter, the software is finally released to the customer or user. Sometimes these steps are not strictly followed most especially when the software need urgent changes possibly due to operating system upgrade, business changes that require quick response due to product competition, if a serious faults has to be corrected. This process could also have adverse effect on the software such as complexity and quick aging. As soon as the software is released, the iteration process continues to ensure the next release of the software product. The circle continues like that and new versions of the software will continue to be released as long as the software is in use.

4. CONCLUSION

Evolution is a critical issue in the life cycle of all software systems. The process of software evolution is driven by requests for changes and includes change impact analysis, release planning, and change implementation. The resultant evolution appears to be driven and controlled by human decision, managerial ethics, and programmer judgement. In this paper, we have discussed the process of improving the quality and maintainability of software through software evolution. We discuss software evolution, evolution process, how evolution permits building better concepts, methods, and tools to assist developers and maintainers, techniques and approaches for monitoring, planning, and predicting successful evolutionary paths for long lived software projects. Finally, we argue that software evolution helps to improve software quality and provides easy and better maintainability for future software maintainers.

REFERENCES

1. Ogheneovo, E. E. (2013). Software Maintenance and Evolution: The Implication for Software Development. West African Journal of Industrial and Academic Research, Vol. 7, No. 2, pp. 34-42.
2. Lehman and Belady (1985). Program Evolution: Processes of Software Change, London: Academic Press.
3. Lehman, M. M. and Ramil, J. F. (2003). Software Evolution: Background, Theory, Practice. Information Process. Lett., 88(1-2), pp. 33-44.
4. Lehman, M. M. and Ramil, J. F. and Kahen, G. (2000). Evolution as a Noun and Evolution as a Verb. In Proceedings of the Workshop on Software and Organization Co-evolution (SOCE2000), Imperial College, London, July 2000.
5. Riaz, M., Mendes, E. and Tempero, E. D. (2009). A Systematic Review of Software Maintainability Prediction and Metrics. In Proceedings of the ESEM 2009, pp. 367-377.
6. Riaz, M., Mendes, E. and Tempero, E. D. (2009). Towards Predicting Maintainability for Relational Database-Driven Software Applications: Extended Evidence from Software Practitioners, Int'l Journal of Software Engineering and Its Applications, Vol. 5, No. 2, April 2011, pp. 107-121.
7. Lehman, M. M. (1980). Programs, Life Cycles, and Laws of Software Evolution. In Proceedings of the IEEE, Vol. 68, No. 9, pp. 1060-1076.
8. IEEE Standard 610.12-1990: Standard Glossary of Software Engineering Technology, IEEE Computer Society Press, Los Alamitos, CA, 1993.
9. ISO/IEC-9126 standard, ISO/IEC 9126-1: Software Engineering—Product Quality—Part I: Quality Model, ISO Standard, Geneva, Switzerland, 2001.
10. ISPSE, 1998. International Workshop on the Principles of Software Evolution, Kyoto, Japan, 1998.
11. ISPSE, 2000. International Workshop on the Principles of Software Evolution, Kanazawa, Japan, November, 2000.
12. Ramil, J. F. and Lehman, M. M. (2000). Metrics of Software Evolution as Effort Predictors –A Case Study. In Proceedings Int'l Conference on Software Maintenance, ICSM 2000, IEEE Computer Society, San Jose, CA, pp. 163-172.
13. Ogheneovo, E. E. (2014a). Software Dysfunction: Why Do Software Fail? Journal of Computer and Communications, 2(6), 25-35. http://dx.doi.org/10.4236/jcc.2014.26004.
14. Ogheneovo, E. E. (2014b). On the Relationship between Software Complexity and Maintenance Costs. Journal of Computer and Communications, 2, 1-16. http://dx.doi.org/10.4236/jcc.2014.214001.
15. Ramil, J. F. and Lehman, M. M. (2000). Cost Estimation and Evolvability Monitoring for Software Evolution Processes. In Workshop on Empirical Studies of Software (WESS'2000), Oct. 14, 2000, San Jose, CA, pp. 1-7.
16. Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., and Tan, W.-G. (1999). Types of Software Evolution and Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 2, pp. 3-30.
17. Lehman, M. M. and Ramil, J. F. (2000). Software Evolution in the Age of Component-Based Software Engineering. IEEE Proceedings on Software Engineering, Vol. 147, pp. 249-255.
18. Bach, J. (2003). The Challenge of "Good Enough" Software, 2003. http://www.satisfice.com/articles/gooden2.pdf. Retrieved 10/10/2018.
19. Denning, W. E. (1988). Out of the Crisis: Quality, Productivity and Competitive Position, Cambridge University Press.
20. Foster, W. A. (). Quality Quote Perpetuates Idiotic Mistake. http://tortschnekkel.blogspot.com/2009/05/willa-foster-quality-quote-perpetuates.html.
21. Krasner, H. (1998). Key Dimensions of Software Quality, ISO 9000-3, IEEE SE Standards.
22. Ivan, I. and C. Boja (2007), Practice for Software Application Optimization, ASE Printing House, Bucharest, pp. 483, 2007.
23. Hashim, K. and Key, E. (1996). A Software Maintainability Attributes Model. Malaysian Journal of Computer Science, Vol. 9, No. 2, December 1996, pp. 92-97.
24. Coenen, F. and Bench-Capon, T. (1993). Maintenance of knowledge-Based Systems: Theory, Techniques and Tools, Hartnolls Ltd., Bodmin, Cornwall, UK.
25. ISO/IEC-9126 standard (ISO/IEC 9126-1) Software Engineering—Product Quality—Part II: Quality Model, ISO Standard, Geneva, Switzerland, 2004.
26. IEEE STD. 610.12
27. Bhatt, P., K. Williams, G. Shroff and A. K.Misra (2006). Influencing Factors in Outsources Software Maintenance, ACM SIGSOFT SEN, 31, 3, 2006, pp. 1-6.
28. Pressman, R. S. (2005). Software Engineering: A Practitioner's Approach, 6th Edition, Boston, McGraw-Hill.

29. Granja-Alvarez and Barranco-Garcia (1997). A Method for Estimating Maintenance Cost in a Software Project: A Case Study, Journal of Software Maintenance Research and Practice, Vol. 9, pp. 161-175.
30. Sommerville, I. (2007). Software Engineering 8th Edition, New York, Addison-Wesley.
31. Dash, Y. Dubey, S. K. Rana, A. (2012). Maintainability Prediction of Object-Oriented Software System by Using Artificial Neural Network Approach. Int'l Journal of Soft Computing and Engineering (IJSCE), ISSN: 2231-2307, Vol. 2, Issue 2, May 2012, pp. 420-423.
32. Saini, R., Dubey, S. K. and Rana, A. (2011). Analytical Study of Maintainability Models for Quality Evaluation. Indian Journal of Computer Science and Engineering (IJCSE), ISSN: 0976-5166, Vol. 2, No. 3, June-July, 2011, pp. 449-454.
33. Goel, N., Dubey, S. K., Rana, A. (2012). Fuzzy Layered Approach for Maintainability Evaluation of Object-Oriented Software System, Int'l Journal of Scientific & Engineering Research, Vol. 3, Issue 6, June 2012, ISSN 2229-5518, pp. 1-8.
34. Welker, K. D. and Oman, F. W. (1995). Software Maintainability Metrics Models in Practice. Journal of defense Software Engineering, Vol. 8, No. 11, November/December, 1995, pp. 19-23.
35. Polo, M., Piattini, M., Riaz, F. (2001). Using Code Metric to Predict Maintenance of Legacy Programs: A Case Study. Proceedings of the Int'l Conference on Software Maintenance, ICSM'01, IEEE Computer Society, Florence, Italy, 2001, pp. 202-208.