Software Maintenance and Evolution: The Implication for Software Development

Ogheneovo, Edward Erhieyovwe.
Department of Computer Science, University of Port Harcourt, Nigeria.
edward_ogheneovo@yahoo.com

Abstract

Software maintenance is the process of modifying existing operational software by correcting errors, migration of the software to new technologies and platforms, and adapting it to deal with new environmental requirements. It denotes any change made to a software product before and after delivery to customer or user. Software maintenance is an important activity of many of organizations today. This is no surprise given the rate of hardware obsolescence, the immortality of a software product, and the demand of users to ensure that existing software products run on newer platforms, run on newer environments, and or with enhanced features. Software maintenance forms an essential component of software development. Therefore, with an increasing use of computers in almost every organization whether big or small, there has emerged the need for software maintenance. In this paper, we argue that software maintenance and evolution are characterized by huge costs, slow speed of implementation, increased complexity, requires technical expertise, be in line with new technologies, may introduce new faults, yet changes and improvements are inevitable if software must stand the test of time.

Keywords: Software, software maintenance, software evolution, reverse engineering,

1.0 Introduction

Software maintenance is an essential component of software development process [25]. It is the process of modifying existing operational software by way of correcting errors, migration of the software to new technologies and platforms, and adapting it to deal with new environmental requirements. As software grows in size, it becomes necessary to determine the complexity of such software [2]. The size of software increases as the complexity increases [18]. Software maintenance is the general process of changing a system after it has been delivered. As noted by [26], software evolution is important because organizations are now completely dependent on their software systems and have invested millions of dollars in these systems. Their systems are critical business assets and they must invest in system change to maintain the value of these assets. Thus, the majority of the software budget in large companies is devoted to maintaining existing systems. Erlikh [5] suggest that about 90% of software costs are evolution costs. Although this percentage may not be exactly correct, but the fact remains that the large chunk of software costs is expended on software maintenance.

As the 21st century advances, more than 50% of the global software population is engaged in modifying existing applications rather than writing new applications. As noted by [9], this should not be a surprise because as an industry grows in size, capacity, and aging, the number of personnel who does repair often outnumbers the number of personnel who build new products. This is very common in almost all area of human endeavour be it automobile, software, etc. at the end of the 20th century, software maintenance grew rapidly especially the year culminating the Y2K. This was largely due to need for updates in software to be able to meet the required modifications needed for software that meets about 85% of the world's supply of existing software application. So two mass update were required to ensure that these software are not obsolete. The first was the set of changes needed to support the new unified European currency (euro) which came into existence in January 1999. Due to the introduction of the Euro currency, about 10% of the world software needed modifications to meet this new currency and also, in the European Monetary Union, about 50% of the information systems required modification in order to support the euro. Secondly, there was the Y2K problem which affected about 75% of the installed software applications operating throughout the world. The Y2K problem also affected not just these software but also some embedded computers inside physical devices such as medical equipments, telephone switching systems, oil wells, and electric generating plants. With software maintenance, these two problems were taking care of although with some penalties such as time wastage as a result of the delays that was triggered in other kinds of software and also in other organizations that depended directly and indirectly in these products. Therefore, concerns for Y2K compliance emphasize the need for understanding and improving the management of software activities.

With an increasing use of computers and software, there has emerged a need for software maintenance in almost every organization [25]. Therefore, software must evolve and be maintained over time [22] and the only way this can be done is to maintain them if they must meet the current and even new challenges. Technology is changing on daily basis and software cannot be left behind if they must be in line with new technologies. It is no longer new that on daily basis, several new communication and security gadgets are being produced and new features are added to them. With these new features, existing software must be modified and new changes must be made to them in order to keep pace with these new technologies and stand the test of time.

2.0 Background of Software Maintenance and Evolution

Maintenance is generally used to describe all changes made to a program after its first installation. It differs significantly from restoration of a system or systems component to its former state [12]. Software evolution plays an important role in software engineering. In fact, most development effort and expenditure is allocated to the evolution and update of these existing versions. Software is ceaselessly changed – maintained, evolved and updated – more often than it is written, and changing software is extremely costly [16]. Normally, the software systems become more and more complex as the evolution and updates of the software systems. Such increasing complexity confronts much more challenges in system robustness and adaptability, which depends on predetermined factor to ensure that long-term safety and reduce the cost of maintenance. Maintenance plays an important role in the life cycle of a software product. It is estimated that there are more than one billion lines of code in production in the world. As much as 80% of it is unstructured, patched and not well documented. It is through the process of maintenance that these problems are alleviated.

2.1 What is Software Maintenance?

Software products must satisfy user's requirements. Software products must change or evolve. As software products are used, faults must be discovered, operating environments may change, and new user requirements will surface. In fact, most of

the maintenance processes are often requested by software user(s). Although the maintenance phase of a software life cycle commences after delivery, however, maintenance activities occur much earlier. Maintenance is one of the primary life cycle processes.

Software maintenance is defined in the IEEE Standard and Software Maintenance, IEEE 1219, as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment (IEEE STD 1219, 1998). According to the ISO/IEC 12207 Standard for Life Cycle Processes, maintenance is the process of a software product undergoing modification to code and associated documentation due to a problem or the need for improvement. According to them, the main objective is to modify existing software product while preserving its integrity. However, a more generally accepted definition of software maintenance by researcher and practitioners is that provided by SWEBOK. It defines software maintenance as totality of activities required to provide cost effective support to a software system. Based on this definition, activities are performed during the pre-delivery and post-delivery stages. Pre-delivery activities include: planning for post-delivery stage, supportability, and logistics while post-delivery activities include modification, training, and operating a help desk [23].

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 [8]. Granja-Alvarez and Barranco-Garcia [7] 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.

2.1.1 Characteristics of Software Maintenance and Evolution

As software ages, it becomes increasingly difficult to keep them up and running without maintenance. Thus there is need to maintain software to keep it is good shape in order to be able to meet new challenges. Some characteristics of software that affect maintenance are system size, age, slow in implementation, support new technology, and structure.

- **Size:** As software is being maintained, it continues to grow in size and as a result becomes more and more complex. Just imagine software with a few hundred lines of code designed to handle a particular application, such software will not be very complex. Consider software such as an operating system with several millions of lines of code (LOC) and developed by about a thousand software engineers. There is no doubt that such software will be very complex due to its large size. As a result, maintaining large software will be very difficult and cumbersome than maintaining small size software.
- **Age:** Software must evolve over time. That is, every software product continues to evolve after its development through maintenance efforts. As the age of software increases, so also the software depreciates. So in order to ensure that the software remain useful and valid, there is the need to maintain it.

- **Slow in Implementation:** Software maintenance takes time. This is so because the maintainer may not be the person who first developed the software. So before the maintainer can begin to maintain the software, there is the need to properly study the software using the program documentation if it exists or try to study the software for some time before the maintenance process can begin. This will in no doubt take some time and effort which will slow down implementation process in an organization
- **Support New Technology:** For software to evolve and support new technology, it must be properly maintained. This is to ensure that it meets the requirements of users of the new system. This way, the software will have to be properly redesigned and possibly re-engineered.
- **Structure:**
  The structure of software system does affect its maintenance especially if the software is not well structured as at the time it was initially developed. Software system developed using spaghetti code will give the maintainer problems in trying to change it to a structured program that can easily be understood by future maintainer.

2.1.2 Types of Software Maintenance

Software must evolve over time. That is, every software product continues to evolve after its development through maintenance efforts. ISO/IEC 14764 and IEEE Computer Society IEEE 1219 classify software maintenance as: corrective, adaptive, perfective, and preventive maintenance.

- **Corrective Maintenance:** This form of maintenance involves the software product after delivery to correct errors e. g., fixing of bugs. Usually, in corrective maintenance, errors are corrected and the cost of corrective maintenance is bored by the developer.
- **Adaptive Maintenance:** Adaptive maintenance involves modifications to the software as a result of changes in operating environments. This is because software must adapt to its environment after delivery. These operating environments could result as a result of database upgrades, operating system upgrade, changes in compiler version, etc. Usually, the organization requesting for adaptive maintenance usually bear the costs.
- **Perfective Maintenance:** Perfective maintenance often involves changes to the code to allow the software to meet the same requirements but in a significantly more acceptable
- manner. This form of maintenance is usually carried out after the software product has been delivered to improve performance.
- **Preventive Maintenance:** In preventive maintenance, software usually modified after delivery to detect and correct latent faults before they become effective faults that can now affect the result of users.

2.1.3 The Nature of Software Maintenance

Software maintenance forms an essential component of software development. Its planning includes estimation of maintenance effort, personnel and costs [25]. Maintenance has a very broad scope. In encompasses a lot of changes both for tracking and controlling software. It is generally used to describe all changes to a program after its first installation. It differs significantly from restoration of a system or system component too its former state [12] [10]. Software maintenance involves the modification of a software product after delivery to correct faults, to improve performance or other attributes. When we talk about software maintenance, people do think that it is merely the process of fixing bugs. This is a wrong notion. Software maintenance involves much more, it is not just about fixing bugs. Studies have shown that over 80% of the maintenance effort is expended on non-corrective actions [23].

Software maintenance and evolution of systems was first proposed by Lehman in 1969. Lehman notes that systems continue to evolve over time. As a result, they become more complex unless some actions such as code refactoring is adopted to reduce the complexity that may arise as a result of maintenance. Software maintenance is a very broad activity that includes error correction, enhancements of capabilities, removal of obsolete functions, and optimization. Because changes are inevitable, certain mechanisms must be developed to evaluate, control, and modify the software. Thus changes to software are done in order to preserve the value of the software over time. The software value can be enhanced by expanding the customer base, meeting additional requirements, thus making the software more easier to use, more efficient, and employing new technology to cater for the new features that may be introduced to these technologies.

The software maintenance process can last for years or even decades after development process [11]. Therefore, there is need for effective planning in order to address the scope of software maintenance, the tailoring of the post delivery/deployment, the designation of who will provide maintenance, and an estimation of the life-cycle costs. Software maintenance takes more effort than all other phases of software life cycle. As noted in [1], about 60 to 70% effort is expended on maintenance phase of software development life cycle. Thus software maintenance activities span a system's productive life cycle and consume a major portion of the total life cycle costs of the system. However, what is actually done to maintenance is sometimes a mystery to many organizations. Thus software maintenance remains an opaque activity that is expensive and difficult to manage.

2.2 What is Software Evolution?

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 as by adding new functionalities and making the existing codes more structured and providing better documentations for users to be able to use such software. 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. [3] 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 the activities and processes.

Therefore, it suffixes to say that software evolution is a change process of software. This change process concerns both hardware and software starting from its development up to the retirement time of the system, during which the system changes into a different and usually more complex or better state [28]. System evolution is part of the system life cycle. Thus the resultant evolution of software appears to be driven and controlled by human decision, managerial edict, and programmer judgment [12]). Software *evolution* takes place only 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, where more accurate requirements are based on the past experience with the application. Businesswise, software is being evolved because the software is successful in the marketplace, revenue accrues from it is high, user demand is strong, the development atmosphere is vibrant and positive, and the organization is supportive, and return on investment is excellent.

Stammel et al. [28] identified several reasons for evolving software. These include:

- New requirements to the system
- Change of Environment
- Evolution of the technology stack especially in the case of strong coupling to it which could result in the co-evolution of certain parts of the software

As noted by [28] the ability to evolve software rapidly and reliably by preserving the architectural integrity of an application is a challenge for every organization. Thus the evolution of a software system may become very costly and expensive resulting in what is referred to as software erosion especially in legacy system due to exposure to changes in processes [20]. By software erosion we mean s the decreasing quality of the internal structure of a software system. This could occur at early development stages of the system. Software erosion may be for example caused by:

- Unmanaged or unstructured introduction of new features (processing of change requests or bugs)
- Unmanaged or unstructured changes of the system
- Unclear or outdated system architecture or bad system development, e.g., software redundancy through copy and paste programming" or violation of architectural decisions
- Loss of knowledge about the system by team fluctuation, or insufficient or outdated documentation

2.2.1 Lehman's Laws of Software Evolution

Lehman [12] and Lehman and Belady [14] examined the growth and evolution of a number of large software projects and from these they were able to propose a number of laws which they called Lehman's laws. The first five laws were those originally proposed by Lehman in his paper titled "Programs, Life Cycles, and Laws of software Evolution" [12]. After further works, three additional laws were proposed [14]. The studies which began with three laws in 1980 have given rise to eight laws of software evolution as formulated and refined by Lehman and his colleagues. These laws seek to consistently account for observed phenomena regarding the evolution software releases, systems and E-Type applications [24]. These laws are summarized below.

i). **Continue change:** According to this law, system maintenance is an inevitable process. As the system's environment changes, new requirements must emerge and the system must be modified. That is, any software system used in the real-world must change or become less and less useful in that environment. Therefore, software must undergo continual change or it becomes progressively less useful.

ii). **Increasing Complexity:** As software undergo changes, they become increasingly more and more complex. To avoid this situation, extra effort and resources must be put in place to ensure that the structure of the system is maintained.

iii). **Large Program Evolution:** Program evolution is a self-regulating process and measurements of system attributes such as size, time between releases, number of reported errors, etc., reveal statistically significant trends and invariances for each system that is released.

iv). **Organizational stability:** During the lifetime of a program, the rate of development of that program is approximately constant and independent of the resources devoted
to system development.

v). **Conservation of Familiarity:** Throughout the duration of a software system, the incremental system change in each release is approximately constant.

vi). **continuing Growth:** As software evolve, there is the need for continual growth to ensure its usefulness. That is, the functionality offered by any software must continually increase for user's satisfaction.

vii). **Declining Quality:** For a system to maintain its quality there is need for it to be adapted to changes in its operational environment otherwise it is bound to decline.

viii). **Feedback System:** Since processes involve multi-agent, multi-loop feedback, they must be treated as such to ensure significant improvement. For Lehman, the software development process itself should be the place to focus attention on, this is what Lehman viewed as feedback-driven and biased toward increasing complexity.

2.2.2 Problems Inherent in Software Maintenance

Software maintenance work is very expensive and takes more time than required. The reasons adduced for this are not farfetched. First, maintenance work is mostly carried out using ad hoc techniques because less attention is paid to software maintenance and as a result fire-brigade approach is often used in software maintenance instead of systematic and planned techniques. Secondly, software maintenance has a very poor image in industries and the result is that competent engineers are not often employed to carry out maintenance. Since maintenance is more challenging than development, there is therefore the need to fully understand the software by the maintainer before necessary modifications and extensions are carried out. Also, since most products are legacy products, they are usually difficult to maintain. This is because legacy systems are often poorly documented, unstructured (usually spaghetti codes with poor and ugly control structures), and lacking in expert(s) knowledgeable in such software products. Although legacy software are believed to be "aged" software developed a long time ago, however, a recently developed software having poor design and documentation can be considered a legacy system.

Another major source of concern is the high costs of maintaining software. It has been estimated that a large chunk of software costs is devoted to maintaining the software. It is often the most expensive and laborious phase especially in legacy codes. Software maintenance alone costs about 70-80% of total costs of software development process. Therefore, in order to ensure that the software stand the test of time, be able to adapt to different environments, and be able to persist for a long time, it must be maintained.

Sometimes maintenance is difficult when the developers are no longer available [30]. Maintenance must take the products of the development e.g., code, documentation, and evolve/maintain them over the life cycle. Therefore, for software that is not properly documented, it becomes a tedious and herculean task to maintain them without properly understanding them. For a maintainer to maintain software product, enough time must be devoted to understanding it. Also, problem of maintainer turnover, recruitment of experience maintainers, maintenance bid costing and time span, and optimization of resource allocation have made long term estimation of maintainer costs a challenging and daunting task to organizations.

2.3 Software Reverse Engineering

Reverse engineering is becoming since legacy software products lack proper documentation, and are highly unstructured. The output of a reverse engineering activity is synthesized, higher-level information that enables the reverse engineer to reason about the system and to evolve it in an effective way. The process of reverse engineering usually focuses on carrying out changes to the code to improve its readability, structure and understanding, without changing any of its functionalities. Reverse engineering is taking apart an object to see how it works in order to duplicate or enhance the object to make it more functional of perform better [6]. Reverse engineering often involves taking software or program apart and analyzing its working in detail, usually to try to make a new device or program that does the same thing without copying anything from the original. The process of reverse engineering usually starts with lower levels of information such as the system's source code, which may also include the system's build environment.

Therefore, reverse engineering is the process of analyzing s system in order to create representations of the system at a higher level of abstraction. It is also seen as the process of going backward through the development cycle [31]. Therefore, reverse engineering is the way of analyzing a system to identify its current components and their dependencies, and to extract and create system abstractions and design information [4]. Through the process of reverse engineering, a system is analyzed to identify its components and their interrelationships and to create the representations of the system in another form or a higher level of abstraction and to create the physical representation of that system.

Software reverse engineering is the process of reversing the design and the requirement specification of a product from an analysis of its code the purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system [17]. It is concerned with the analysis (not modification) of an existing (software) system [19]. The IEEE Standard for Software Maintenance (IEEE Std 1219-1993) defines reverse engineering as the process of extracting software system information (including documentation) from source code. It involves reversing a program's machine code back into the source code that it was written in, using programming language statements. Usually, software reverse engineering is done in order to retrieve the source code of a program because the source code is lost. It is used to study how the program performs certain operations. As noted by [6], software reverse engineering is done to improve the performance of a program and fixing of bugs. It is often used to identify malicious content in a program such as virus.

Usually, before the process of reverse engineering, some cosmetic changes to the code are usually carried out. This is done to improve its readability, structure and understandability, without changing any of its functionality. The cosmetic changes are carried out using these steps:

- **Reformat program:** The first step is to reformat the legacy software to make the program look neat and appeal to the eyes of the reader. Reformatting the legacy code will help to remove complex control structures
- **Assigning Variable Names:** Variable names that are meaningful are then assigned to enable the reader comprehend the program faster. This will provide good information for code documentation. Therefore, all variables, data statements, and functions should be assigned meaningful names whenever possible.

- **Simplify Conditions:** Nested conditions should be simplified in the program as much as possible. Complex nested conditions in the program should be replaced by simpler conditional statements. This is to ensure that complexity is reduced in the software.
- **Remove GO TOs:** The use of Go TOs should be discouraged in the program. If they are already in the legacy code, measure should be put in place by the software maintainer to remove them and replace them with other control constructs that are appropriate. GO TO statements usually make programs to be complex and often causes an infinite loops.
- **Simplify Processing:** The program
- is the simplified having removed unnecessary nested conditions and GO TO statements that could make the program to be complex.

After these cosmetic changes have been carried out on the legacy code, the process of reverse engineering then starts. This involves extracting the code, design, and the requirement specification then begins. After reverse engineering, the process starts all over again from forward engineering which involve the reversal of the reverse engineering.

2.4 Software Maintenance Models

Different types of models are in use in software maintenance. However, five models are mostly used in the industry. They are: quick fix model, Boehm's model, Osborne's model, the iterative enhancement model, and the reuse oriented model.

- **The Quick Fix Model:** This is an ad hoc model. The goal of this model is to identify a problem and the quickly fix it [29]. As the name implies, the model does not pay attention to the long-term effects as a result of time constraint. However, the quick fix model is fast and it gets work done quickly with lower cost.
- **Boehm's Model:** This model is based on economic models and principles. Boehm economic model helps us to better understand the problem and improve productivity in maintenance. The model provides an overview of economic analysis techniques and their applicability to software engineering and management. This model provides a balanced view of candidate software engineering solutions, and a framework that takes account of both programming and human problems and proffering solution to these problems.
- **Osborne's Model:** This model is concerned with the reality of the maintenance environment. According to Osborne, technical problems that arise during maintenance are due to poor communication and control between management. Osborne therefore recommends four strategies to address these technical problems. These are:

1). **Maintenance requirements** need to be included in the change specification
2). A quality assurance program is required to establish quality assurance requirements.
3). A metric needs to be developed in order to verify that the maintenance goals have been met
4). Managers need to provide with feedback through performance reviews

- **The Iterative Enhancement Model**

This model state that changes made to the system during the lifetime make up an iterative process. Software enhancement implies addition of new features or functionalities to an existing application. The iterative model has three stages: 1) the system has to be analyzed, 2) the proposed modifications are then classified, and 3) the changes are then implemented. However, for the iterative enhancement model to work, the system must be completely documented since the model assumes that a full documentation of the system exists before enhancement can begin.

- **The Reuse Oriented Model:** This model assumes that existing program components could be reused. Software products are very costly. In order to reduce costs, parts of previously developed software can be reused. In addition to reduce development costs and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality. Component-based software development is a good example of a reuse approach [21]. This approach ensures that software product is developed through off-the-shelf components. Therefore, the steps for the reuse model are identifying the parts of the old system which have the potential for reuse, understanding the system parts, modifying the old or existing system based on new requirements, and integrating the modified parts into the new system.

Conclusion

Software maintenance is the process of modifying existing operational software by correcting errors, migration of the software to new technologies and platforms, and adapting it to deal with new environmental requirements. It denotes any change made to a software product before and after delivery to customer or user.

Software maintenance is an important activity of many of organizations today.

This is no surprise given the rate of hardware obsolescence, the immortality of a software product, and the demand of users to ensure that existing software products run on newer platforms, run on newer environments, and or with enhanced features. Software maintenance forms an essential component of software development. As software grows in size, it becomes necessary to determine the complexity of such software. The size of software increases as the complexity increases. Software maintenance is the general process of changing a system after it has been delivered. In this paper, we argue that software maintenance and evolution are characterized by huge costs, slow speed of implementation, increased complexity, requires technical expertise, be in line with new technologies, may introduce new faults, yet changes and improvements are inevitable if software must stand the test of time.

References

[1] Barry, E. J., Kemerer, C. F. and Slaughter, S. A. (1999). Toward a Detailed Classification Scheme for Software Maintenance Activities. In Proceedings of the 5th Americas Conference on Information Systems, August 1999 pp. 126-128.

[2] Bhattacherjee, V., Kumar, P. and Kumar, M. S. (2009). Complexity Estimation, Journal of Theoretical and Applied Metric for Analogy Based Effort-Information Technology, Vol. 6, No. 1, pp. 1-8.

[3] 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.

[4] Chikofsky, E. J. and Cross, J. H. (1990). Reverse Engineering and Design Recovery: Taxonomy in IEEE Software, IEEE Computer Society, pp. 13-17.

[5] Erlikh, L. (2000). Leveraging Legacy System Dollars for E-Business. In Information Technology Proceedings, May/June 2000, pp. 17-23.

[6] Garg, M. and Jindal, M. K. (2009). Reverse Engineering – Roadmap to Effective Software Design, Int'l Journal of Recent Trends in Engineering, Vol. 1, N0. 2, pp. 186-188.

[7] Granja-Alvarez, J. C. and Barranco-Garcia, M. J. (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.

[8] Frost, D. (1990). Software Maintenance and Modification. In Proceedings of Software Maintenance and Computers, San Diego, CA, IEEE Computer Society Press Tutorial, Los Alamitos, CA, pp. 187-192.

[9] Jones, C. (2006). The Economics of Software Maintenance in the Twenty First Century, Version 3, Software Productivity Research, Inc., http://www.spr.com

[10] Kemerer, C. F. (1997). Software Project Management Readings and Cases, Mcgraw-Hill Companies, Inc, Irwin Book Team, pp. 510-520.

[11] Kemerer, C. F. and Slaughter, S. A. (1999). An Empirical Approach to Studying Software Evolution, IEEE Transactions of Software Engineering, IEEE Transactions on Software Engineering, Vol. 25, No. 4, July/August 1999, pp. 493-509.

[12] Lehman, M. M. (1980). Programs, Life Cycles, and Laws of Software Evolution. In Proceedings of the IEEE, Vol. 68, No. 9, pp. 1060-1076.

[13] Lehman, M. M. (1985). Program Evolution, London: Academia Press

[14] Lehman and Belady (1985). Program Evolution: Processes of Software Change, London: Academic Press.

[15 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.

[16] Li, H., Huang, B. and Lii, J. (2008). Dynamic Evolution Analysis of Object-Oriented Software Systems, IEEE Congress on Evolution Computation (CEC 2008), pp. 3035-3040.

[17] Mall, R. (2009). Fundamentals of Software Engineering (3rd Edition), PHI Learning

Private Ltd., New Delhi, India, pp. 404-411.

[18] Mishra, A. and Misra, S. (2010). People Management in Software Industry: The Key to Success, Journal of ACM Sigsoft Software Engineering Notes, Vol. 35, No. 6, pp. 1-4.

[19] Müller, H. A. and Kienle, H. M. (2010). Encyclopedia of Software Engineering, Taylor & Francis, chapter Reverse Engineering, pp. 1016–1030. http://www.tandfonline.com/doi/abs/10.1081/E-ESE-120044308.

[20] Niessink, F. and Van Vliet, H. (2000). Software Maintenance from a Service Perspective. Journal of Software Maintenance and Evolution: Research and Practice 12, pp. 103-120.

[21] Ning, J. Q. (1996). A Component-Based Software Development Model, In Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC'96), pp.389–394, IEEE.

[22] Pfleeger, S. L. (1998). Software Engineering –Theory and Practice, Prentice Hall

[23] Pigosky, T. M. (2001). Software Maintenance. IEEE –Trial Version 1.00 –May 2001.

[24] Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S. and Lakhani, K. (2006). Understanding Open Source Software Evolution in Madhavji, N. H. Lehman, M. M., Ramil, J. F. and Perry, D. (eds.), Software Evolution and Feedback, John Wiley and sons Inc, New York, 2006.

[25] Shukla, R. and Misra, A. K. (2008). Estimating Software Maintenance Effort –A Neural Network approach. In Proceedings of the Int'l Software Engineering Conference, February 19-20, 2008, Hyderabad, India, pp. 107-112.

[26] Sommerville, I. (2007). Software Engineering, 8th Ed., Addison-Wesley, New York, NY, Software Evolution, pp. 488-511.

[27] Slonneger, K. (2004). Software Development: An Introduction

[28] Stammel, J. Durdik, Z., Krogmann, K. Weiss, K. and Koziolek, H. (2001). Software Evolution for Industrial automation Systems: Literature Overview, Karlsruhe Institute of Technology, University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association.

[29] Takang, A. A. and Grubb, P. A. (1996). Software Maintenance Concepts and Practices. Thompson Computer Press London, UK.

[30] Torchiano, M., Ficca, F., De Lucia, A. (2007). Empirical Studies in Software Maintenance and Evolution. IEEE Int'l Conference on Software Maintenance, ICSM'07, pp. 491-494.

[31] Warden, R. (1992). Software Reuse and Reverse Engineering in Practice, London, Chapman & Hall, pp. 283-305.