International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 03 Issue 08
April 2016

Software Maintenance

Bandana Kumari
B.Tech. Scholar, Indus Institute of Technology & Management Bilhaur, Kanpur Dr. A.P.J.Abdul Kalam Technical University
vandana9589@gmail.com

Abstract
This paper overviews software maintenance, its relevance, the problems, and the available solutions; the underlying objective is to present software maintenance not as a problem, but in terms of solutions. Of course, this view of maintenance does not apply to software, as software does not deteriorate with the use and the passing of time. Nevertheless, the need for modifying a piece of software after delivery has been with us since the very beginning of electronic computing. The Lehman's laws of evolution [17, 18] state that successful software systems are condemned to change over time. A predominant proportion of changes is to meet ever changing user needs. This is captured by the first law of Lehman [17, 18]: "A program that is used in a real world environment necessarily must change or become progressively less useful in that environment". Significant changes also derive from the need to adapt software to interact with external entities, including people, organizations, and artificial systems. In fact, software is infinitely malleable and, therefore, it is often perceived as the easiest part to change in a system [6].
Keywords: Corrective maintenance; Adaptive maintenance; Perfective maintenance; Emergency maintenance; iterative-enhancement

1 Introduction
The term maintenance, when accompanied to software, assumes a meaning profoundly different from the meaning it assumes in any other engineering discipline. In fact, many engineering disciplines intend maintenance as the process of keeping something in working order, in repair. The key concept is the deterioration of an engineering artifact due to the use and the passing of time; the aim of maintenance is therefore to keep the artifact's functionality in line with that defined and registered at the time of release.

2 Definitions
Software maintenance is a very broad activity often defined as including all work made on a software system after it becomes operational [21]. This covers the correction of errors, the enhancement, deletion and addition of capabilities, the adaptation to changes in data requirements and operation environments, the improvement of performance, usability, or any other quality attribute. The IEEE definition is as follows [11]:

"*Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.*"

This definition reflects the common view that software maintenance is a post-delivery activity: it starts when a system is released to the customer or user and encompasses all activities that keep the system operational and meet the user's needs. This view is well summarized by the classical waterfall models of the software life cycle, which generally comprise a final phase of operation and maintenance. Pigoski [23] captures the needs to begin maintenance when development begins in a new definition:

"*Software maintenance is the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage. Pre-delivery activities include planning for post delivery operations, supportability, and logistics determination. Post-delivery activities include software modification, training, and operating a help desk.*"

This definition is consistent with the approach to software maintenance taken by ISO in its standard on software life cycle processes [15]. It definitively dispels the image that software maintenance is all about fixing bugs or mistakes.

3 Categories of software maintenance

ISO [14] introduces three categories of software maintenance:

Problem resolution, which involves the detection, analysis, and correction of software nonconformities causing operational problems;

Interface modifications, required when additions or changes are made to the hardware system controlled by the software;

Functional expansion or performance improvement, which may be required by the purchaser in the maintenance stage.

The IEEE definition of maintainability reflects the definition of maintenance: 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 [11]. ISO assumes maintainability as one of the six primary characteristics of its definition of software quality and suggests that it depends on four sub-characteristics: analyzability, changeability, stability, testability [13]; the new version of the standard, currently under development, adds compliance as a fifth sub-characteristic.

A recommendation is that all changes should be made in accordance with the same procedures, as far as possible, used for the development of software. However, when resolving problems, it is possible to use temporary fixes to minimize downtime, and implement permanent changes later.

IEEE [12] redefines the Lientz and Swanson [20] categories of *corrective*, *adaptive*, and *perfective* maintenance, and adds *emergency maintenance* as a fourth category. The IEEE definitions are as follows [12]:

*"**Corrective maintenance**: reactive modification of a software product performed after delivery to correct discovered faults.*

***Adaptive maintenance**: modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.*

***Perfective maintenance**: modification of a software product performed after delivery to improve performance or maintainability.*

***Emergency maintenance**: unscheduled corrective maintenance performed to keep a system operational."*

These definitions introduce the idea that software maintenance can be either scheduled or unscheduled and reactive or proactive.

4 Costs and challenges

However one decides to categorize the maintenance effort, it is still clear that software maintenance accounts for a huge amount of the overall software budget for an information system organization. Since 1972 [7], software maintenance was characterized as an "iceberg" to highlight the enormous mass of potential problems and costs that lie under the surface.

Several technical and managerial problems contribute to the costs of software maintenance. Among the most challenging problems of software maintenance are: *program comprehension*, *impact analysis*, and *regression testing*.

One of the major challenges in software maintenance is to determine the effects of a proposed modification on the rest of the system. Once a change has been implemented, the software system has to be retested to gain confidence that it will perform according to the (possibly modified) specification. The process of testing a system after it has been modified is called regression testing [19]. The aim of regression testing is twofold: to establish confidence that changes are correct and to ensure that unchanged portions of the system have not been affected. Regression testing differs from the testing performed during development because a set of test cases may be available for reuse. Indeed, changes made during a maintenance process are usually small (major rewriting are a rather rare event in the history of a system) and, therefore, the simple approach of executing all test cases after each change may be excessively costly. Alternatively, several strategies for selective regression testing are available that attempt to select a subset of the available test cases without affecting test effectiveness [10, 24].

5 Models

A typical approach to software maintenance is to work on code first, and then making the necessary changes to the accompanying documentation, if any. Ideally, after the code has been changed the requirement, design, testing and any other form of available documents impacted by the modification should be updated. However, due to its perceived malleability, users expect software to be modified quickly and cost-effectively. Changes are often made on the fly, without proper planning, design, impact analysis, and regression testing. Documents may or may not be updated as the code is modified; time and budget pressure often entails that changes made to a program are not documented and this quickly degrades documentation. In addition, repeated changes may demolish the original design, thus making future modifications progressively more expensive to carry out.

Evolutionary life cycle models suggest an alternative approach to software maintenance. These models share the idea that the requirements of a system cannot be gathered and fully understood initially. Accordingly, systems are to be developed in builds each of which completes, corrects, and refines the requirements of the previous builds based on the feedback of users [9]. An example is iterative enhancement [2], which suggests structuring a problem to ease the design and implementation of successively larger/refined solutions. The construction of a new build (that is, maintenance) begins with the analysis of the existing system's requirements, design, and code and test documentation and continues with the modification of the highest-level document affected by changes, propagating the changes down to the full set of documents. In short, at each step of the evolutionary process the system is redesigned based on an analysis of the existing system.

A key advantage of the iterative-enhancement model is that documentation is kept updated as the code changes. Visaggio [26] reports data from replicated controlled-experiments conducted to compare the quick-fix and the iterative-enhancement models and shows that the maintainability of a system degrades faster with the quick-fix model. The experiments also indicate that organizations adopting the iterative-enhancement model make maintenance changes faster than those applying the quick-fix model; the latter finding is counter-intuitive, as the most common reason for adopting the quick-fix model is time pressure.

The iterative-enhancement model is well suited for systems that have a long life and evolve over time; it supports the evolution of the system in such a way to ease future modifications. On the contrary, the full-reuse model is more suited for the development of lines of related products. It tends to be more costly on the short run, whereas the advantages may be sensible in the long run; organizations that apply the full-reuse model accumulate reusable components of all kinds and at many different levels of abstractions and this makes future developments more cost effective.

6 Processes

6.1 **Reverse engineering**

Reverse engineering as been defined as "the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction" [8]. Accordingly, reverse engineering is a process of examination, not a process of change, and therefore it does not Involve changing the software under examination.

The IEEE Standard for Software Maintenance [12] suggests that the process of reverse engineering evolves though six steps: dissection of source code into formal units; semantic description of formal units and creation of functional units; description of links for each unit (input/output schematics of units); creation of a map of all units and successions of consecutively connected units (linear circuits); declaration and semantic description of system applications, and; creation of an anatomy of the system. The first three steps concern local analysis on a unit level (in the small), while the other three steps are for global analysis on a system level (in the large).

Benedusi et al. [5] advocate the need for a high-level organizational paradigm when setting up complex processes in a field, such as reverse engineering, in which methodologies and tools are not stable but continuously growing. The role of such a paradigm is not only to define a framework in which available methods and tools can be used, but also to allow the repetitions of processes and hence to learn from them. They propose a paradigm, called Goals/Models/Tools, that divides

the setting up of a reverse engineering process into the following three sequential phases: Goals, Models, and Tools.

Goals: this is the phase in which the motivations for setting up the process are analyzed so as to identify the information needs and the abstractions to be produced. Models: this is the phase in which the abstractions identified in the previous phase are analyzed so as to define representation models that capture the information needed for their production.

Tools: this is the phase for defining, acquiring, enhancing, integrating, or constructing: extraction tools and procedures, for the extraction from the system's artifacts of the row data required for instantiating the models defined in the model phase; and abstraction tools and procedures, for the transformation of the program models into the abstractions identified in the goal phase.

The Goals/Models/Tools paradigm has been extensively used to define and execute several real-world reverse engineering processes [4, 5].

6.2 Re-engineering

The practice of re-engineering a software system to better understand and maintain it has long been accepted within the software maintenance community. Chikofsky and Cross II taxonomy paper [8] defines re-engineering as "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form". The same paper indicates renovation and reclamation as possible synonyms; renewal is anothercommonly used term. Arnold [1] gives a more comprehensive definition as follows:

"Software Re-engineering is any activity that: (1) improves one's understanding of software, or (2) prepares or improves the software itself, usually for increased maintainability, reusability, or evolvability."

it is evident hat re-engineering entails some form of reverse engineering to create a more abstract view of a system, a regeneration of this abstract view followed by forward engineering activities to realize the system in the new form. The presence of a reverse engineering step distinguishes re-engineering from restructuring, the latter consisting of transforming an artifact from one form to another at the same relative level of abstraction [8]. Software re-engineering has proven important for several reasons. Arnold [1] identifies seven main reasons that demonstrate the relevance of re-engineering:

"Re-engineering can help reduce an organization's evolution risk;
Re-engineering can help an organization recoup its investment in software;
Re-engineering can make software easier to change;
Re-engineering is a big business;
Re-engineering capability extends CASE toolsets;
Re-engineering is a catalyst for automatic software maintenance;
Re-engineering is a catalyst for applying artificial intelligence techniques to solve software re-engineering problems."

7 Maintenance management

Management is "the process of designing and maintaining an environment in which individuals, working together in groups, accomplish efficiently selected aims" [27]. In the case of maintenance the key aim is to provide cost-effective support to a software system during its entire lifespan. Management is concerned with quality and productivity that imply effectiveness and efficiency. Many authors [16, 27, 25] agree that management consists of five separate functions. The functions are: planning, organizing, staffing, leading (sometimes also called directing), and controlling.

Planning consists of selecting missions and objectives and predetermining a course of actions for accomplishing them. Commitment of human and material resources and scheduling of actions are among the most critical activities in this function.

Organizing is the management function that establishes an intentional structure of roles for people to fill in an organization. This entails arranging the relationships among roles and granting the responsibilities and needed authority.

Staffing involves filling the positions in the organization by selecting and training people. Two key activities of this function are evaluating and appraising project personnel and providing for

general development, i.e. improvement of knowledge, attitudes, and skills.

Leading is creating a working environment and an atmosphere that will assist and motivate people so that they will contribute to the achievement of organization and group goals.

Controlling measures actual performances against planned goals and, in case of deviations, devises corrective actions. This entails rewarding and disciplining project personnel.

The standard IEEE-1219 [12] suggests a template to guide the preparation of a software maintenance plan based on the standard itself; figure. Pigoski [23] highlights that a particular care must be made to plan the transition of a system from the development team to the maintenance organization, as this is a very critical element of the life cycle of a system.

Software maintenance organizations can be designed and set up with three different organizational structures: functional, project, or matrix [25, 28].

Functional organizations are hierarchical in nature. The maintenance organization is broken down into different functional units, such as software modification, testing, documentation, quality assurance, etc. Functional organizations present the advantage of a centralized organization of similar specialized resources. The main weakness is that interface problems may be difficult to solve: whenever a functional department is involved in more than a project conflicts may arise over the relative priorities of these projects in the competition for resources. In addition, the lack of a central point of complete responsibility and authority for the project may entails that a functional department places more emphasis on its own specialty than on the goal of the project.

Project organizations are the opposite of the functional organizations. In this case a manager is given the full responsibility and authority for conducting the project; all the resources needed for accomplishing the project goals are separated from the regular functional structure and organized into an autonomous, self-contained team. The project manager may possibly acquire additional resources from outside the overall organization. Advantages of this type of organization are a full control over the project, quick decision making, and a high motivation of project personnel. Weaknesses include the fact that there is a start-up time for forming the team, and there may be an inefficient use of resources.

Matrix organizations are a composition of functional and project organizations with the objective of maximizing the strengths and minimizing the weaknesses of both types of organizations. The standard vertical hierarchical organization is combined with a horizontal organization for each project. The strongest point of this organization is that a balance is struck between the objectives of the functional departments and those of the projects. The main problem is that every person responds to two managers, and this can be a source of conflicts. A solution consists of specifying the roles, responsibility and authority of the functional and project managers for each type of decisions to be made.

A common problem of software maintenance organizations is inexperienced personnel. Beath and Swanson [3] report that 25% of the people doing maintenance are students and up to 61% are new hires. Pigoski [23] confirms that 60% to 80% of the maintenance staff is newly hired personnel. Maintenance is still perceived by many organizations as a non strategic issue, and this explain why it is staffed with students and new hired people. To compound the problem there is the fact that most Universities do not teach software maintenance, and maintenance is very rarely though in corporate training and education programs, too. As an example, software maintenance is not listed within the 22 software courses of the software engineering curriculum sketched in reference [22]. The lack of appraisal of maintenance personnel generates other managerial problems, primarily high turnover and low morale.

8 Conclusions

This article has overviewed software maintenance, its strategic problems, and the available solutions. The underlying theme of the article has been to show that technical and managerial solutions exist that can support the application of high standards of engineering in the maintenance of software. Of course, there are open problems and more basic and applied research is needed both to gain a better

understanding of software maintenance and to find better solutions.

Nowadays, the way in which software systems are designed and built is changing profoundly, and this will surely have a major impact on tomorrow's software maintenance. Object technology, commercial-off-the-shelf products, computer supported cooperative work, outsourcing and remote maintenance, Internet/Intranet enabled systems and infrastructures, user enhance able systems, are a few examples of areas that will impact software maintenance. Object technology has become increasingly popular in recent years and a majority of the new systems are currently being developed with an object-oriented approach. Among the main reasons for using object technology is enhanced modifiability, and hence easier.

References

[1] Arnold, R. S., "A Road Map to Software Re-engineering Technology", Software Reengineering - a tutorial, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 3-22.

[2] Basili, V. R., "Viewing Maintenance as Reuse-Oriented Software Development", IEEE Software, 7(1):19-25, 1990.

[3] Beath, C. N., Swanson, E. B., "Maintaining Information Systems in Organizations", John Wiley & Sons, New York, NY, 1989.

[4] Benedusi, P., Cimitile, A., De Carlini, U., "A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams", Proceedings of the Conference on Software Maintenance, Miami, FL, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 180-189.

[5] Benedusi, P., Cimitile, A., De Carlini, U., "Reverse Engineering Processes, Document Production and Structure Charts", The Journal of Systems and Software, 16:225-245, 1992.

[6] Brooks, F. P. Jr., "No Silver Bullet", IEEE Computer, 20(4):10-19, 1987.

[7] Canning, R., "The Maintenance Iceberg", EDP Analyzer, 10(10), 1972.

[8] Chikofsky, E. J., Cross II, J. H., "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, 7(1):13-17, 1990.

[9] Gilb, T., "Principles of Software Engineering Management", Addison-Wesley, Reading, MA, 1988.

[10] Hartmann, J., Robson, D. J., "Techniques for Selective Revalidation", IEEE Software, 16(1):31-38, 1990.

[11] IEEE Std. 610.12, "Standard Glossary of Software Engineering Terminology", IEEE Computer Society Press, Los Alamitos, CA, 1990.

[12] IEEE Std. 1219-1998, "Standard for Software Maintenance", IEEE Computer Society Press, Los Alamitos, CA, 1998.

[13] ISO/IEC 9126, "Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for Their Use", Geneva, Switzerland, 1991.20

[14] ISO/IEC 9000-3, "Quality Management and Quality Assurance Standards – Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software", Geneva, Switzerland, 1991.

[15] ISO/IEC 12207, "Information Technology – Software Life Cycle Processes", Geneva, Switzerland, 1995.

[16] Koontz, H., O'Donnell, C., "Principles of Management: An Analysis of Managerial Functions", fifth edition, McGraw-Hill, New York, NY, 1972.

[17] Lehman, M. M., "Lifecycles and the Laws of Software Evolution", Proceedings of the IEEE, Special Issue on Software Engineering, 19:1060-1076, 1980.

[18] Lehman, M. M., "Program Evolution", Journal of Information Processing Management, 19(1):19-36, 1984.

[19] Leung, H. K. N., White, L. J., "Insights into Regression Testing", Proceedings of the Conference on Software Maintenance, Miami, Florida, IEEE Computer Society Press, 1990, pp. 60-69.

[20] Lientz, B. P., Swanson, B. E., "Software Maintenance Management", Addison- Wesley, Reading, MA, 1980.

[21] Martin, J., Mc Clure, C., "Software Maintenance – the Problem and its Solutions", Prentice Hall, Englewood Cliffs, NJ, 1983.

[22] Parnas, D. L., "Software Engineering Programs are not Computer Science Programs", IEEE Software, 16(6):19-30, 1999.

[23] Pigoski, T. M., "Practical Software Maintenance – Best Practices for Managing Your Software Investment", John Wiley & Sons, New York, NY, 1997.

[24] Rothermel, G., Harrold, M. J., "A Framework for Evaluating Regression Test Selection Techniques", Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, IEEE Computer Society Press, CA, 1994, pp. 201-210.

[25] Thayer, R. H., "Software Engineering Project Management", Software Engineering Project Management, Second Edition, Thayer, R. H., ed., IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 72-104.

[26] Visaggio, G., "Assessing the Maintenance Process through Replicated Controlled Experiments", The Journal of Systems and Software, 44(3):187-197, 1999.

[27] Weihrich, H., "Management: Science, Theory, and Practice", Software Engineering Project Management, Second Edition, Thayer, R. H., ed., IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 4-13.

[28] Youker, R., "Organization Alternatives for Project Managers", Project Management Quarterly, 8(1), The Project Management Institute, 1997.