Software Maintenance

Article · January 2001

DOI: 10.1142/9789812389718_0005 · Source: CiteSeer

2 authors:

Gerardo Canfora
Università degli Studi del Sannio
345 PUBLICATIONS **12,212** CITATIONS

SEE PROFILE

Aniello Cimitile
Università degli Studi del Sannio
170 PUBLICATIONS **2,545** CITATIONS

SEE PROFILE

Software Maintenance

Gerardo Canfora and Aniello Cimitile
gerardo.canfora@unisannio.it, cimitile@unisannio.it

University of Sannio, Faculty of Engineering at Benevento
Palazzo Bosco Lucarelli, Piazza Roma
82100, Benevento Italy

Tel. ++39 0824 305804 Fax. ++39 0824 21866

29 November, 2000

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.

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 [50, 51] 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 [50, 51]: "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 [21].

This article 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.

The remainder of the article is organized as follows. Section 2 defines software maintenance and section 3 categorizes it. Costs and challenges of software maintenance are analyzed in section 4. Sections 5-7 are devoted to the structure of the maintenance activity. In particular, section 5 introduces general models of the maintenance process, while section 6 discusses existing standards. Management of software maintenance is discussed in section 7. Sections 8

and 9 deal with two related areas that support software maintenance, namely reverse engineering and reengineering. Section 10 is devoted to legacy systems, an issue that assumed an ever increasing economic relevance in the last decade. Concluding remarks and areas for further investigation are given in section 11, while section 12 provides resources for further reading.

2 Definitions

Software maintenance is a very broad activity often defined as including all work made on a software system after it becomes operational [56]. 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 [40]:

> "*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, as shown in figure 1.

Several authors disagree with this view and affirm that software maintenance should start well before a system becomes operational. Schneidewind [67] states that the myopic view that maintenance is strictly a post-delivery activity is one of the reasons that make maintenance hard. Osborne and Chikofsky [59] affirm that it is essential to adopt a life cycle approach to managing and changing software systems, one which looks at all aspects of the development process with an eye toward maintenance. Pigoski [62] 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 [44]. It definitively dispels the image that software maintenance is all about fixing bugs or mistakes.

3 Categories of software maintenance

Across the 70's and the 80's, several authors have studied the maintenance phenomenon with the aim of identifying the reasons that originate the needs for changes and their relative frequencies and costs. As a result of these studies, several classifications of maintenance activities have been defined; these classifications help to better understand the great significance of maintenance and its implications on the cost and the quality of the systems in use. Dividing the maintenance effort into categories has first made evident that software maintenance is more than correcting errors.

Lientz and Swanson [54] divide maintenance into three components: *corrective*, *adaptive*, and *perfective* maintenance. Corrective maintenance includes all the changes made to remove actual faults in the software. Adaptive maintenance encompasses the changes needed as a consequence of some mutation in the environment in which the system must operate, for instance, altering a system to make it running on a new hardware platform, operating system, DBMS, TP monitor, or network. Finally, perfective maintenance refers to changes that originate from user requests; examples include inserting, deleting, extending, and modifying functions, rewriting documentation, improving performances, or improving ease of use. Pigoski [62] suggests joining the adaptive and perfective categories and calling them *enhancements*, as these types of changes are not corrective in nature: they are improvements. As a matter of fact, some organizations use the term software maintenance to refer to the implementation of small changes, whereas software development is used to refer to all other modifications.

Ideally, maintenance operations should not degrade the reliability and the structure of the subject system, neither they should degrade its maintainability[1], otherwise future changes will be progressively more difficult and costly to implement. Unfortunately, this is not the case for real-world maintenance, which often induces a phenomenon of aging of the subject system [60]; this is expressed by the second law of Lehman [50, 51]: "As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure". Accordingly, several authors consider a fourth category of maintenance, named *preventive* maintenance, which includes all the modifications made to a piece of software to make it more maintainable [63].

ISO [43] 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. A

[1] 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 [40]. 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 [42]; the new version of the standard, currently under development, adds compliance as a fifth sub-characteristic.

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 [41] redefines the Lientz and Swanson [54] categories of *corrective*, *adaptive*, and *perfective* maintenance, and adds *emergency maintenance* as a fourth category. The IEEE definitions are as follows [41]:

> *"**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, as shown in figure 2. Figure 3 depicts the correspondences that exist between ISO and IEEE categories.

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 [28], software maintenance was characterized as an "iceberg" to highlight the enormous mass of potential problems and costs that lie under the surface. Although figures vary, several surveys [1, 2, 7, 10, 34, 46, 54, 56, 58] indicate that software maintenance consumes 60% to 80% of the total life cycle costs; these surveys also report that maintenance costs are largely due to enhancements (often 75–80%), rather than corrections.

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*.

Whenever a change is made to a piece of software, it is important that the maintainer gains a complete understanding of the structure, behavior and functionality of the system being modified. It is on the basis of this understanding that modification proposals to accomplish the maintenance objectives can be generated. As a consequence, maintainers spend a large amount of their time reading the code and the accompanying documentation to comprehend its logic, purpose, and structure. Available estimates indicate that the percentage of maintenance time consumed on program comprehension ranges from 50% up to 90% [32, 55, 73]. Program comprehension is frequently compounded because the maintainer is rarely the

author of the code (or a significant period of time has elapsed between development and maintenance) and a complete, up-to-date documentation is even more rarely available [26].

One of the major challenges in software maintenance is to determine the effects of a proposed modification on the rest of the system. Impact analysis [6, 64, 35, 81] is the activity of assessing the potential effects of a change with the aim of minimizing unexpected side effects. The task involves assessing the appropriateness of a proposed modification and evaluating the risks associated with its implementation, including estimates of the effects on resources, effort and scheduling. It also involves the identification of the system's parts that need to be modified as a consequence of the proposed modification. Of note is that although impact analysis plays a central role within the maintenance process, there is no agreement about its definition and the IEEE Glossary of Software Engineering Terminology [40] does not give a definition of impact analysis.

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 [52]. 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 [39, 66].

Most problems that are associated with software maintenance can be traced to deficiencies of the software development process. Sneidewind [67] affirms that "the main problem in doing maintenance is that we cannot do maintenance on a system which was not designed for maintenance". However, there are also essential difficulties, i.e. intrinsic characteristics of software and its production process, that contribute to make software maintenance an unequalled challenge. Brooks [21] identifies complexity, conformity, changeability, and invisibility as four essential difficulties of software and Rajlich [65] adds discontinuity to this list.

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. This approach is captured by the quick-fix model, shown in figure 4, which demonstrates the flow of changes from the old to the new version of the system [8]. 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 [36]. An example is iterative enhancement [8], which suggests structuring a problem to ease the design and implementation of successively larger/refined solutions. Iterative enhancement explains maintenance too, as shown in figure 5. The construction of a new build (that is, maintenance) begins with the analysis of the existing system's requirements, design, 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 [76] 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.

Basili [8] suggests a model, the full-reuse model shown in figure 6, that views maintenance as a particular case of reuse-oriented software development. Full-reuse begins with the requirement analysis and design of a new system and reuses the appropriate requirements, design, code, and tests from earlier versions of the existing system. This is a major difference with the iterative enhancement, which starts with the analysis of the existing system. Central to the full-reuse model is the idea of a repository of documents and components defining earlier versions of the current system and other systems in the same application domain. This makes reuse explicit and documented. It also promotes the development of more reusable components.

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

Several authors have proposed process models for software maintenance. These models organize maintenance into a sequence of related activities, or phases, and define the order in which these phases are to be executed. Sometimes, they also suggest the deliverables that each phase must provide to the following phases. An example of such a process is shown in figure 7 [82]. Although different authors identify different phases, they agree that there is a core set of activity that are indispensable for successful maintenance, namely the comprehension of the existing system, the assessment of the impact of a proposed change, and the regression testing.

IEEE and ISO have both addressed software maintenance, the first with a specific standard [41] and the latter as a part of its standard on life cycle processes [44]. The next two sections describe the maintenance processes defined by these two documents.

6.1 IEEE-1219

The IEEE standard organizes the maintenance process in seven phases, as demonstrated in figure 8. In addition to identifying the phases and their order of execution, for each phase the standard indicates input and output deliverables, the activities grouped, related and supporting processes, the control, and a set of metrics.

Problem/modification identification, classification, and prioritization. This is the phase in which the request for change (MR – modification request) issued by a user, a customer, a programmer, or a manager is assigned a maintenance category (see section 3 for maintenance categories definitions), a priority and a unique identifier. The phase also includes activities to determine whether to accept or reject the request and to assign it to a batch of modifications scheduled for implementation.

Analysis. This phase devises a preliminary plan for design, implementation, test, and delivery. Analysis is conducted at two levels: feasibility analysis and detailed analysis. Feasibility analysis identifies alternative solutions and assess their impacts and costs, whereas detailed analysis defines the requirements for the modification, devises a test strategy, and develop an implementation plan.

Design. The modification to the system is actually designed in this phase. This entails using all current system and project documentation, existing software and databases, and the output of the analysis phase. Activities include the identification of affected software modules, the modification of software module documentation, the creation of test cases for the new design, and the identification of regression tests.

Implementation. This phase includes the activities of coding and unit testing, integration of the modified code, integration and regression testing, risk analysis, and review. The phase also includes a test-readiness review to asses preparedness fort system and regression testing.

Regression/system testing. This is the phase in which the entire system is tested to ensure compliance to the original requirements plus the modifications. In addition to functional and

interface testing, the phase includes regression testing to validate that no new faults have been added. Finally, this phase is responsible for verifying preparedness for acceptance testing.

Acceptance testing. This level of testing is concerned with the fully integrated system and involves users, customers, or a third party designated by the customer. Acceptance testing comprises functional tests, interoperability tests, and regression tests.

Delivery. This is the phase in which the modified systems is released for installation and operation. It includes the activity of notifying the user community, performing installation and training, and preparing and archival version for backup.

6.2 ISO-12207

While the standard IEEE-1219 [41] is specifically concerned with software maintenance, the standard ISO-12207 [44] deals with the totality of the processes comprised in the software life cycle. The standard identifies seventeen processes grouped into three broad classes: primary, supporting, and organizational processes. Processes are divided into constituent activities each of which is further organized in tasks. Figure 9 shows the processes and their distribution into classes. Maintenance is one of the five primary processes, i.e. one of the processes that provide for conducting major functions during the life cycle and initiate and exploit support and organizational processes. Figure 10 shows the activities of the maintenance processes; the positions do not indicate any-time dependent relationships.

Process implementation. This activity includes the tasks for developing plans and procedures for software maintenance, creating procedures for receiving, recording, and tracking maintenance requests, and establishing an organizational interface with the configuration management process. Process implementation begins early in the system life cycle; Pigoski [62] affirms that maintenance plans should be prepared in parallel with the development plans. The activity entails the definition of the scope of maintenance and the identification and analysis of alternatives, including offloading to a third party; it also comprises organizing and staffing the maintenance team and assigning responsibilities and resources.

Problem and modification analysis. The first task of this activity is concerned with the analysis of the maintenance request, either a problem report or a modification request, to classify it, to determine its scope in term of size, costs, and time required, and to assess its criticality. It is recommended that the maintenance organization replicates the problem or verifies the request. The other tasks regard the development and the documentation of alternatives for change implementation and the approval of the selected option as specified in the contract.

Modification implementation. This activity entails the identification of the items that need to be modified and the invocation of the development process to actually implement the changes. Additional requirements of the development process are concerned with testing procedures to ensure that the new/modified requirements are completely and correctly implemented and the original unmodified requirements are not affected.

Maintenance review/acceptance. The tasks of this activity are devoted to assessing the integrity of the modified system and end when the maintenance organization obtain the approval for the satisfactory completion of the maintenance request. Several supporting processes may be invoked, including the quality assurance process, the verification process, the validation process, and the joint review process.

Migration. This activity happens when software systems are moved from one environment to another. It is required that migration plans be developed and the users/customers of the system be given visibility of them, the reasons why the old environment is no longer supported, and a description of the new environment and its date of availability. Other tasks are concerned with the parallel operations of the new and old environment and the post-operation review to assess the impact of moving to the new environment.

Software retirement. The last maintenance activity consists of retiring a software system and requires the development of a retirement plan and its notification to users.

7 Maintenance management

Management is "the process of designing and maintaining an environment in which individuals, working together in groups, accomplish efficiently selected aims" [79]. 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 [48, 79, 74] agree that management consists of five separate functions, as shown in figure 11. 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 [41] suggests a template to guide the preparation of a software maintenance plan based on the standard itself; figure 12 shows an outline of this template.

Pigoski [62] 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 [74, 83].

Functional organizations are hierarchical in nature, as shown in figure 13. 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 (see figure 14). 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. Figure 15 shows a matrix organization; the standard vertical hierarchical organization is combined with an 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, as shown in figure 16.

A common problem of software maintenance organizations is inexperienced personnel. Beath and Swanson [10] report that 25% of the people doing maintenance are students and up to 61% are new hires. Pigoski [62] 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 [61]. The lack of appraisal of maintenance personnel generates other managerial problems, primarily high turnover and low morale.

8 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” [29]. Accordingly, reverse engineering is a process of examination, not a process of change, and therefore it does not involve changing the software under examination.

Although software reverse engineering originated in software maintenance, it is applicable to many problem areas. Chikofsky and Cross II [29] identify six key objectives of reverse engineering: coping with complexity, generating alternate views, recovering lost information, detecting side effects, synthesizing higher abstractions, and facilitating reuse. The standard IEEE-1219 [41] recommends reverse engineering as a key supporting technology to deal with systems that have the source code as the only reliable representation. Examples of problem areas where reverse engineering has been successfully applied include identifying reusable assets [23], finding objects in procedural programs [24, 37], discovering architectures [49], deriving conceptual data models [18], detecting duplications [47], transforming binary programs into source code [30], renewing user interfaces [57], parallelizing sequential programs [17], and translating [22], downsizing [70], migrating [27], and wrapping legacy code [72]. Reverse engineering principles have also been applied to business process re-engineering to create a model of an existing enterprise [45].

Reverse engineering as a process is difficult to define in rigorous terms because it is a new and rapidly evolving field. Traditionally, reverse engineering has been viewed as a two step process: information extraction and abstraction. Information extraction analyses the subject system artifacts – primarily the source code – to gather row data, whereas information abstraction creates user-oriented documents and views. Tilley and Paul [75] propose a preliminary step that consists of constructing domain-specific models of the system using conceptual modeling techniques.

The IEEE Standard for Software Maintenance [41] 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. [12] 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 [11, 12].

9 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 [29] 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 another commonly used term. Arnold [5] 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. This process is illustrated in figure 17. 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 [29].

Software re-engineering has proven important for several reasons. Arnold [5] 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."*

Examples of scenarios in which re-engineering has proven useful include migrating a system from one platform to another [19], downsizing [70], translating [22, 31], reducing maintenance costs [69], improving quality [4], and migrating and re-engineering data [3]. The standard IEEE-1219 [41] highlights that re-engineering can not only revitalize a system, but also provide reusable material for future development, including frameworks for object-oriented environments.

Software re-engineering is a complex process that re-engineering tools can only support, not completely automate. There is a good deal of human intervention with any software re-engineering project. Re-engineering tools can provide help in moving a system to a new maintenance environment, for example one based on a repository, but they cannot define such an environment nor the optimal path along which to migrate the system to it. These are activities that only human beings can perform. Another problem that re-engineering tools only marginally tackle is the creation of an adequate testbed to prove that the end product of re-engineering is fully equivalent to the original system. This still involves much hand-checking, partially because very rarely an application is re-engineered without existing functions being changed and new functions being added. Finally, re-engineering tools often fail to take into account the unique aspects of a system, such as the use of a JCL or a TP-Monitor, the accesses to a particular DBMS or the presence of embedded calls to modules in other languages.

Success in software re-engineering requires much more than just buying one or more re-engineering tools. Defining the re-engineering goals and objectives, forming the team and training it, preparing a testbed to validate the re-engineered system, evaluating the degree to which the tools selected can be integrated and identifying the bridge technologies needed, preparing the subject system for re-engineering tools (for example, by stubbing DBMS accesses and calls to assembler routines) are only a few examples of activities that contribute to determining the success of a re-engineering project. Sneed [71] suggests that five steps should be considered when planning a re-engineering project: project justification, which entails determining the degree to which the business value of the system will be enhanced; portfolio analysis, that consists of prioritizing the applications to be re-engineered based on their technical quality and business value; cost estimation, that is the estimation of the costs of the project; cost-benefit analysis, in which costs and expected returns are compared, and; contracting, which entails the identification of tasks and the distribution of effort.

10 Legacy systems

A scenario that highlights the high cost of software maintenance is legacy systems. These are systems developed over the past 20/30 years (or even more) to meet a growing demand for new products and services. They have typically been conceived in a mainframe environment using non-standard development techniques and obsolete programming languages. The structure has often been degraded by a long history of changes and adaptations and neither consistent documentation nor adequate test suites are available. Nevertheless, these are crucial systems to the business they support (most legacy systems hold terabytes of live data) and encapsulate a great deal of knowledge and expertise of the application domain. Sometimes the legacy code is the only place where domain knowledge and business rules are recorded, and this entails that even the development of a new replacement system may have to rely on

knowledge which is encapsulated in the old system. In short, legacy systems have been identified as “large software systems that we don’t know how to cope with but that are vital to our organization” [14]. Similarly, Brodie and Stonebraker [20] define a legacy system as “an information system that significantly resists modifications and evolution to meet new and constantly changing business requirements.”

There are a number of options available to manage legacy systems. Typical solutions include: discarding the legacy system and building a replacement system; freezing the system and using it as a component of a new larger system; carrying on maintaining the system for another period, and; modifying the system to give it another lease of life [15]. Modifications may range from a simplification of the system (reduction of size and complexity) to ordinary preventive maintenance (re-documentation, restructuring and re-engineering) or even to extraordinary processes of adaptive maintenance (interface modification, wrapping and migration). These possibilities are not alternative to each other and the decision on which approach, or combination of approaches, is the most suitable for any particular legacy system must be made based on an assessment of technical and business value of the system.

Four factors that have been successfully used to asses the value of a legacy system and make informed decisions are: obsolescence, deterioration, decomposability, and business value.

Obsolescence measures the ageing of a system due to the progress of software and data engineering and the evolution of hardware/software platforms. Obsolescence induces a cost which results from not taking advantage of modern methods and technologies which reduce the burden of maintaining a system.

Deterioration measures the decreases of the maintainability of a system (lack of analyzability, modifiability, stability, testability, etc.) due to the maintenance operations the system has undergone in its lifespan. Deterioration directly affects the cost of maintaining a system.

Decomposability measures the identifiability and independence of the main components of a system. All systems can be considered as having three components: interfaces, domain functions, and data management services. The decomposability of a system indicates how well these components are reflected in its architecture.

Business Value measures the complexity of the business process and rules a system, or system’s component, implements and their relevance to achieve efficiency and efficacy in the business operation.

Reference [25] suggests how these factors can be measured and introduces a life cycle model for legacy systems that uses these factors to drive decisions. Figure 18 gives an overview of the life cycle model. The model stresses the fact that a system is constantly under maintenance by means of the outermost maintenance loop; the figure also highlights three more loops which refer to ordinary maintenance, extraordinary maintenance, and replacement. Extraordinary maintenance differs from ordinary maintenance for the extent of the modifications and the impact that they have on the underlying business processes. New and replacement systems enter the ordinary maintenance loop; the decisions on whether or not a

running system needs to enter an extraordinary maintenance loop or a replacement loop (decision points D1 and D2) requires that the system has been assessed based on the four factors discussed above. Returning to the ordinary maintenance loop (decision points E1 and E2) entails the evaluation of the progress/completion of the planned process.

Bennett et al. [15] stress the need to model the business strategy of an organization from a top-down perspective, including many stakeholders, to make informed decisions about legacy systems. They introduce a two-phase model, called SABA – Software as a Business Asset, that use an organizational scenario tool to generate scenarios for the organization's future and a technology scenario tool to produce a prioritized set of solutions for the legacy system. Prioritization of solutions is achieved by comparing the current (legacy) system with the systems required by each scenario generated by the organizational scenario tool.

11 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 enhanceable 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 maintenance. This is achieved through concepts such as classes, information hiding, inheritance, polymorphism, and dynamic binding. However, there is no enough data that empirically show the impact of object technology on maintenance [38, 68]. Wilde and Huitt [80] discuss some of the problems that may be expected in the maintenance of software developed with object technology and make recommendations for possible tool support. Among the recognized problems are the fact that inheritance may make the dependencies among classes harder to find and analyze [9, 33] and may cause an increase of rework [53]. Also, single changes may be more difficult to implement with object-oriented software compared to procedural software; however, object-oriented development typically results in fewer changes. In short, these findings suggest that object technology does not necessarily improve maintainability and more empirical studies are needed to understand its impact.

More and more organizations are replacing their in-house systems by acquiring and integrating commercial products and components; the main drivers are quicker time to market and lower development costs. However, commercial-off-the-shelf products have the effect of

reducing the malleability of software and will have a major impact on the maintenance process [77, 78].

As software systems grow in size, complexity and age, their maintenance and evolution cease to be an individual task to require the combined efforts of groups of software engineers. The day-by-day work of these groups of software engineers produces a body of shared knowledge, expertise and experiences, a sort of rationale for the design of the system, that is precious to improve the productivity of the process of evolving the system over the time, and to reduce the chances of introducing errors at each maintenance operation. Whenever this agreed understanding is not available, software engineers have to develop it as a (preliminary) part of their maintenance assignment. This is a costly and time consuming activity: available figures indicate that 2/3 of the time of a software engineer in a maintenance team is spent looking at the code and the available documents to (re-)discover and process information which had probably been already derived several times during system lifetime [13]. Despite of its importance, this knowledge is seldom recorded in any systematic manner; usually, it is in the mind of engineers and is lost when engineers change their job (or duty). Hence, this is a great potential for improvement in the productivity of maintenance teams.

Outsourcing of software maintenance has grown in the past decade and is now a well established industry. The development of telecommunications, and the diffusion of Internet/Intranet infrastructures, are now pushing in the direction of telecommuting and remote maintenance; this will require a rethinking of the way in which maintenance organizations are designed and set up and processes are enacted.

Currently there is a debate on the nature and the essence of software maintenance. Indubitably, the traditional vision of software maintenance as a post-delivery activity dates back to the seventies and is strictly related to the waterfall life cycle models. With the emerging of the iterative and evolutionary life cycle models the fact that maintenance is both a pre-delivery and a post-delivery activity has became apparent. However, there is still now a widespread opinion that software maintenance is all about fixing bugs or mistakes. This is why several authors prefer the term software evolution to refer to non-corrective maintenance. Recently, Bennett and Rajlich [16] have proposed a staged model of the software life cycle that partition the conventional maintenance phase of waterfall models in a more useful way. They retain initial development and add an explicit evolution stage during which functionality and capabilities are extended in a major way to meet user needs. Evolution is followed by a stage of servicing, in which the system is subject to defect repairs and simple changes in functions. Then, the system moves to a phase-out stage and finally to a close-down. The authors claim that evolution is different from servicing, from phase-out, and from close-down, and this difference has important technical and business consequences.

12 Resources

The Journal of Software Maintenance, published by John Wiley & Sons is the only periodical completely dedicated to software maintenance. Articles on software maintenance appear regularly also in The Journal of Systems and Software, published by Elsevier Science. Other journals that deliver software maintenance articles are: the IEEE Transactions on Software

Engineering, the International Journal of Software Engineering and Knowledge Engineering, published by World Scientific, Software Practice and Experience, published by John Wiley & Sons, Information and Software Technology, published by Elsevier Science, Empirical Software Engineering, published by Kluwer Academic Publishers, and the Journal of Automated Software Engineering, published by Kluwer Academic Publishers.

The International Conference on Software Maintenance is the major annual venue in the area of software maintenance and evolution. Other conferences that address the theme of software maintenance are: the Conference on Software Maintenance and Reengineering, the International Workshop on Program Comprehension, the Working Conference on Reverse Engineering, the conference on Software Engineering and Knowledge Engineering, the Workshop on Software Change and Evolution, and the International Conference on Software Engineering.

Pointers for further readings on software maintenance can be found in the Chapter 6 of the Guide to the Software Engineering Body of Knowledge (www.swebok.org), whose purpose is to provide a consensually-validated characterization of the bounds of the software engineering discipline and to provide a topical access to the body of knowledge supporting that discipline. The chapter presents an overview of the knowledge area of software maintenance. Brief descriptions of the topics are provided so that the reader can select the appropriate reference material according to his/her needs.

References

[1] Abran, A., Nguyemkim, H., “Analysis of Maintenance Work Categories Tough Measurement”, Proceedings of the Conference on Software Maintenance, Sorrento, Italy, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 104-113.

[2] Alkhatib, G., “The Maintenance Problem of Application Software: An Empirical Analysis”, Journal of Software Maintenance – Research and Practice, 4(2):83-104, 1992.

[3] Andrusiewicz, A. Berglas, A., Harrison, J., Ming Lim, W., “Evalution of the ITOC Information Systems Design Recovery Tool”, The Journal of Systems and Software, 44(3):229-240, 1999.

[4] Antonini, P., Canfora, G., Cimitile, A., “Re-engineering Legacy Systems to Meet Quality Requirements: An Experience Report”, Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 146-153.

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

[6] Arnold, R. S., Bohner, S. A., “Impact Analysis – Toward a Framework for Comparison”, Proceedings of the Conference on Software Maintenance, Montreal, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 292-301.

[7] Artur, L. J., “Software Evolution: The Software Maintenance Challenge”, John Wiley & Sons, New York, NY, 1988.

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

[9] Basili, V. R., Briand, L. C., Melo, W. L., “A Validation of Object-Oriented Design Metrics as Quality Indicators”, IEEE Transactions on Software Engineering, 22(10):651-661, 1996.

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

[11] 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.

[12] 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.

[13] Bennett, K. H., Younger, E. J., “Model-Based Tools to Record Program Understanding”, Proceedings of the 2nd Workshop on Program Comprehension, Capri, Napoli, Italy, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 87-95.

[14] Bennett, K. H., “Legacy Systems: Coping with Success”, IEEE Software, 12(1):19-23, 1995.

[15] Bennett, K. H., Ramage, M., Munro, M.,“Decision Model for Legacy Systems”, IEE Proceedings on Software, 146(3):153-159, 1999.

[16] Bennett, K. H., Rajlich, V., “The Staged Model of the Software Lifetime: A New Perspective on Software Maintenance”, IEEE Computer, to appear, 2000.

[17] Bhansali, S., Hagemeister, J. R., Raghavendra, C. S., Sivaraman, H., “Parallelizing Sequential Programs by Algorithm-Level Transformations”, Proceedings of the 3rd Workshop on Program Comprehension, Washington, DC, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 100-107.

[18] Blaha, M. R., Premerlani, W. J., “An Approach for Reverse Engineering of Relational Databases”, Communications of the ACM, 37(5):42-49, 1994.

[19] Britcher, R. N., “Re-engineering Software: A Case Study”, IBM System Journal, 29(4):551-567, 1990.

[20] Brodie, M. L., Stonebraker, M.,“Migrating Legacy Systems”, Morgan Kaufmann Publishers, San Mateo, CA, 1995.

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

[22] Byrne, E. J., “Software Reverse Engineering: A Case Study”, Software – Practice and Experience, 21(12):1349-1364, 1991.

[23] Canfora, G., Cimitile, A., Munro, M., “RE^2: Reverse Engineering and Reuse Re-Engineering”, Journal of Software Maintenance – Research and Practice, 6(2):53-72, 1994.

[24] Canfora, G., Cimitile, A., Munro, M., “An Improved Algorithm for Identifying Objects in Code”, Software – Practice and Experience, 26(1):25-48, 1996.

[25] Canfora, G., Cimitile, A., “A Reference Life Cycle for Legacy Systems”, Proceedings of ICSE’97 Workshop on Migration Strategies for Legacy Systems, Technical Report TUV-1841-97-06, Cimitile, A., Muller, H., Klosch, R. R., eds., 1997.

[26] Canfora, G., Cimitile, A., “Program Comprehension”, Encyclopedia of Library and Information Science, volume 66, supplement 29, 1999.

[27] Canfora, G., De Lucia, A., Di Lucca, G. A., “An Incremental Object-Oriented Migration Strategy for RPG legacy Systems”, International Journal of Software Engineering and Knowledge Engineering, 9(1):5-25, 1999.

[28] Canning, R., “The Maintenance Iceberg”, EDP Analyzer, 10(10), 1972.

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

[30] Cifuentes, C., Gough, K. J., “Decompilation of Binary Programs”, Software – Practice and Experience, 25(7):811-829, 1995.

[31] Cifuentes, C., Simon, D., Fraboulet, A., “Assembly to High-Level Language Translation”, Proceedings of the International Conference on Software Maintenance, Bethesda, Maryland, IEEE Computer Society Press, 1998, pp. 228-237.

[32] Corbi, T. A., “Program Understanding: Challenge for the 1990s”, IBM System Journal, 28(2):294-306, 1989.

[33] Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M., “Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software”, Empirical Software Engineering, an International Journal, 1(2):109-132, 1996.

[34] Foster, J. R., “Cost Factors in Software Maintenance”, Ph.D. Thesis, Computer Science Department, University of Durham, Durham, UK, 1993.

[35] Fyson, M. J., Boldyreff, C., “Using Application Understanding to Support Impact Analysis”, Journal of Software Maintenance – Research and Practice, 10(2):93-110, 1998.

[36] Gilb, T., “Principles of Software Engineering Management”, Addison-Wesley, Reading, MA, 1988.

[37] Girard, J. F., Koschke, R.,“A Comparison of Abstract Data Types and Objects Recovery Techniques”, Science of Computer Programming, 36(2-3), 2000.

[38] Glass, R. L., “The Software Research Crisis”, IEEE Software, 11(6):42-47, 1994.

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

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

[41] IEEE Std. 1219-1998, “Standard for Software Maintenance”, IEEE Computer Society Press, Los Alamitos, CA, 1998.

[42] ISO/IEC 9126, “Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for Their Use”, Geneva, Switzerland, 1991.

[43] 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.

[44] ISO/IEC 12207, “Information Technology – Software Life Cycle Processes”, Geneva, Switzerland, 1995.

[45] Jacobson, I., Ericsson, M., Jacobson, A., “The Object Advantage – Business Process Re-engineering with Object Technology”, Addison-Wesley, Reading, MA, 1995.

[46] Jones, C., “Assessment and Control of Software Risks”, Prentice Hall, Englewood Cliffs, NJ, 1994.

[47] Kontogiannis, K., De Mori, R., Merlo, E., Galler, M., Bernstein, M., “Pattern Matching for Clone and Concept Detection”, Journal of Automated Software Engineering, 3:77-108, 1996.

[48] Koontz, H., O’Donnell, C., “Principles of Management: An Analysis of Managerial Functions”, fifth edition, McGraw-Hill, New York, NY, 1972.

[49] Lakhotia, A., “A Unified Framework for Expressing Software Subsystem Classification Techniques”, The Journal of Systems and Software, 36:211-231, 1997.

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

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

[52] 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.

[53] Leung, H. K. N., “The Dark Side of Object-Oriented Software Development”, Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 438.

[54] Lientz, B. P., Swanson, B. E., “Software Maintenance Management”, Addison-Wesley, Reading, MA, 1980.

[55] Livadas, P. E., Small, D. T., “Understanding Code Containing Preprocessor Constructs”, Proceedings of the 3rd Workshop on Program Comprehension, Washington, DC, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 89-97.

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

[57] Merlo, E., Gagne, P.-Y., Girard, J.-F., Kontogiannis, K., Hendren, L., Panangaden, P., De Mori, R., “Re-engineering User Interfaces”, IEEE Software, 12(1):64-73, 1995.

[58] Nosek, J. T., Prashant, P., “Software Maintenance Management: The Changes in the Last Decade”, Journal of Software Maintenance – Research and Practice, 2(3):157-174, 1990.

[59] Osborne, W. M., Chikofsky, E. J., “Fitting Pieces to the Maintenance Puzzle”, IEEE Software, 7(1):11-12, 1990.

[60] Parnas, D. L., “Software Aging”, Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 279-287.

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

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

[63] Pressman, R. S., “Software Engineering – A Practitioner’s Approach”, McGraw-Hill, New York, NY, 1992.

[64] Queille, J. P., Voidrot, J. F., Wilde, N., Munro M. “The Impact Analysis Task in Software Maintenance: A Model and a Case Study”, Proceedings of the International Conference on Software Maintenance, Victoria, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 234-242.

[65] Rajlich, V., “Program Reading and Comprehension”, Proceedings of the Summer School on Engineering of Existing Software, Monopoli, Bari, Italy, Giuseppe Laterza Editore, Bari, Italy, 1994, pp. 161-178.

[66] 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.

[67] Schneidewind, N. F., “The State of Software Maintenance”, IEEE Transactions on Software Engineering, SE-13(3):303-310, 1987.

[68] Slonim, J., “Challenges and Opportunities of Maintaining Object-Oriented Systems”, Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 440-441.

[69] Slovin, M., Malik, S., “Re-engineering to Reduce System Maintenance: A Case Study”, Software Engineering, Research Institute of America, Inc., 1991, pp. 14-24.

[70] Sneed, H., Nyary, E., “Downsizing Large Application Programs”, Proceedings of the International Conference on Software Maintenance, Montreal, Quebec, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 110-119.

[71] Sneed, H., “Planning the Re-engineering of Legcy Systems”, IEEE Software, 12(1):24-34, 1995.

[72] Sneed, H., “Encapsulating Legacy Software for Use in Client/Server Systems”, Proceedings of the 3rd Working Conference on Reverse Engineering, Monterey, CA, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 104-119.

[73] Standish, T. A., “An Essay on Software Reuse”, IEEE Transactions on Software Engineering, SE-10(5):494-497, 1984.

[74] 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.

[75] Tilley, S. R., Paul, S., “Towards a Framework for Program Understanding ”, Proceedings of the 4th Workshop on Program Comprehension, Berlin, Germany, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 19-28.

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

[77] Voas, J., “Are COTS Products and Component Packaging Killing Software Malleability ?”, Proceedings of the International Conference on Software Maintenance, Bethesda, Maryland, IEEE Computer Society Press, 1998, pp. 156-157.

[78] Voas, J., “Maintaining Component-based Systems”, IEEE Software, 15(4):22-27, 1998.

[79] 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.

[80] Wilde, N., Huitt, R., “Maintenance Support for Object-Oriented Programs”, IEEE Transactions on Software Engineering, 18(12):1038-1044, 1992.

[81] Yau, S. S., Colleferro, J. S., “Some Stability Measures for Software Maintenance”, IEEE Transactions on Software Engineering, SE-6(6):545-552, 1980.

[82] Yau, S. S., Nicholl, R. A., Tsai, J. J.P., Liu, S.-S., “An Integrated Life Cycle Model for Software Maintenance”, IEEE Transactions on Software Engineering, SE-14(8):1128-1144, 1988.

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

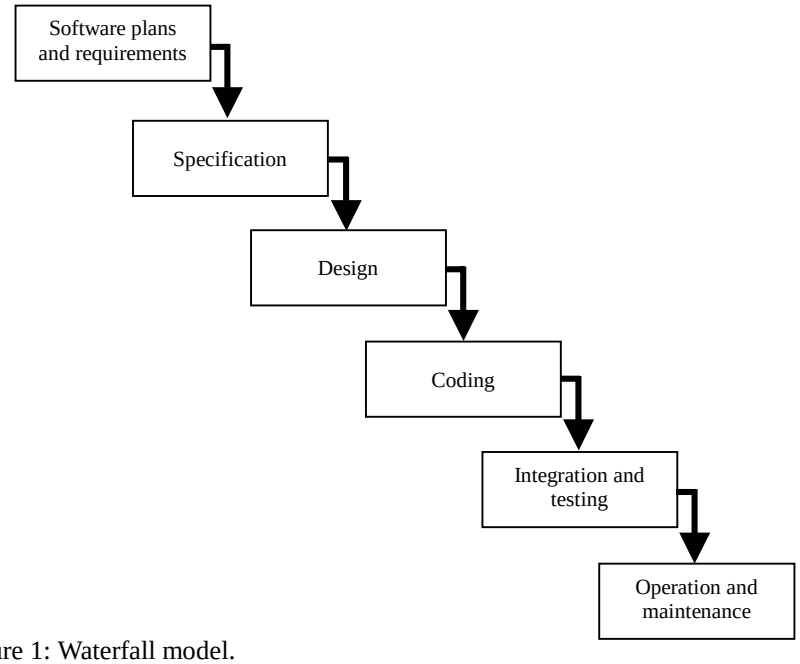

Figure 1: Waterfall model.

	Unscheduled	Scheduled
Reactive	Emergency	Corrective Adaptive
Proactive		Perfective

Figure 2: IEEE categories of software maintenance.

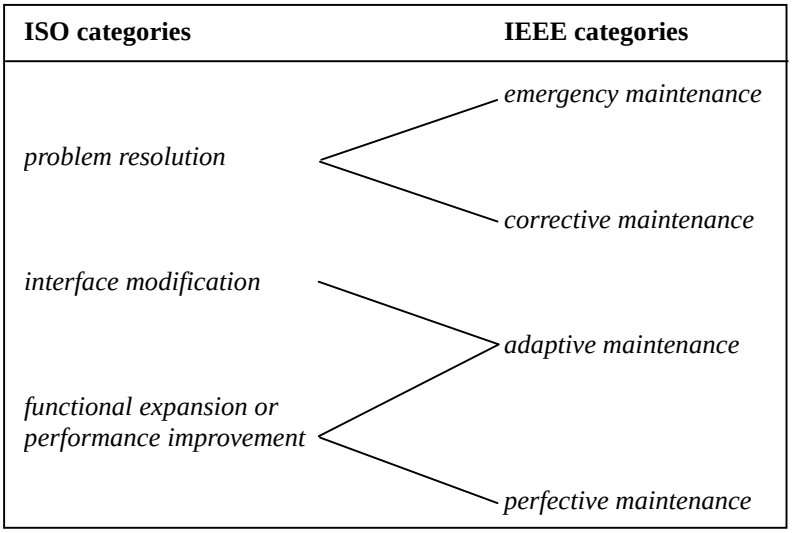

Figure 3: Correspondences between ISO and IEEE maintenance categories.

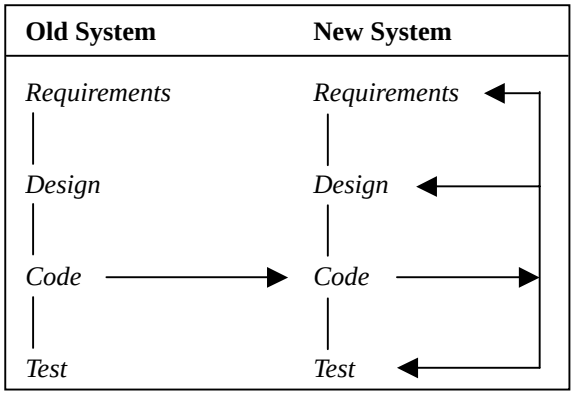

Figure 4: The quick-fix model [8].

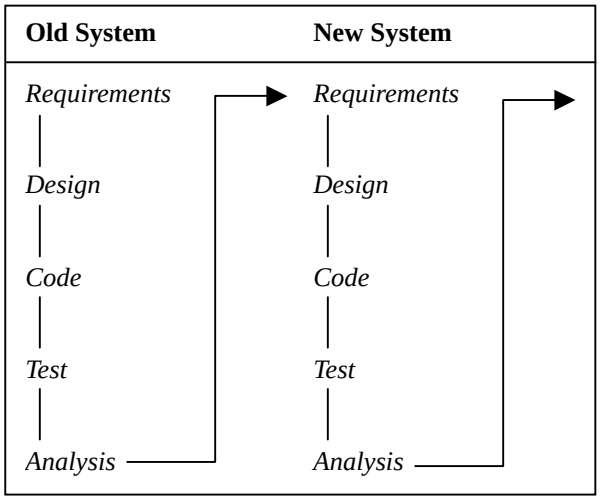

Figure 5: The iterative-enhancement model [8].

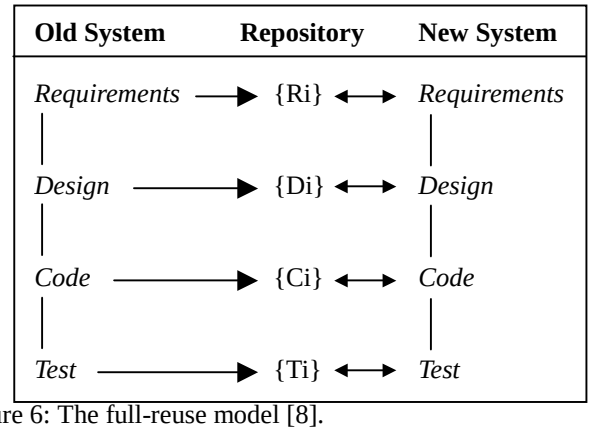

Figure 6: The full-reuse model [8].

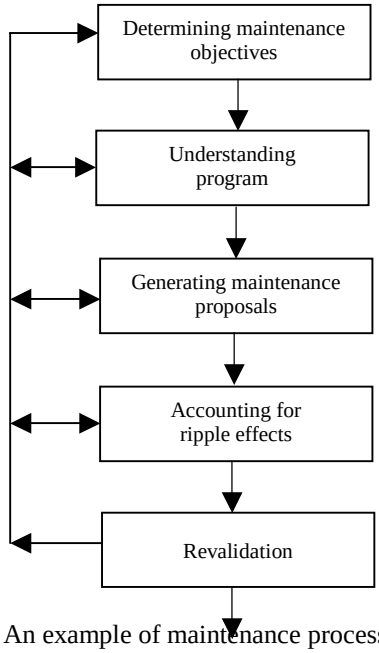

Figure 7: An example of maintenance process [82].

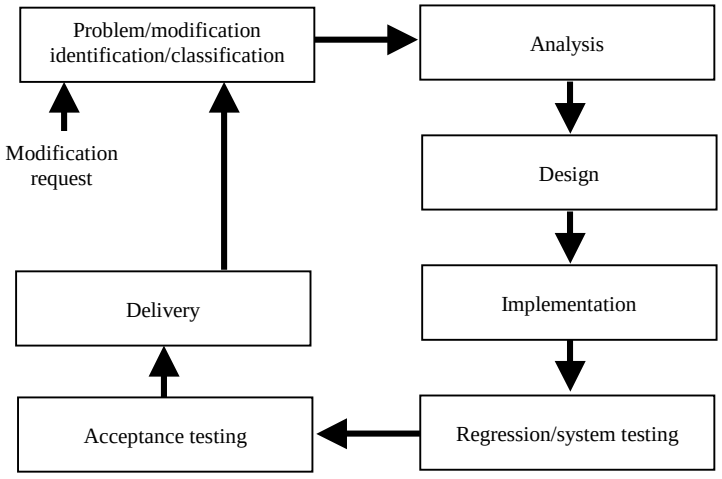

Figure 8: The IEEE maintenance process.

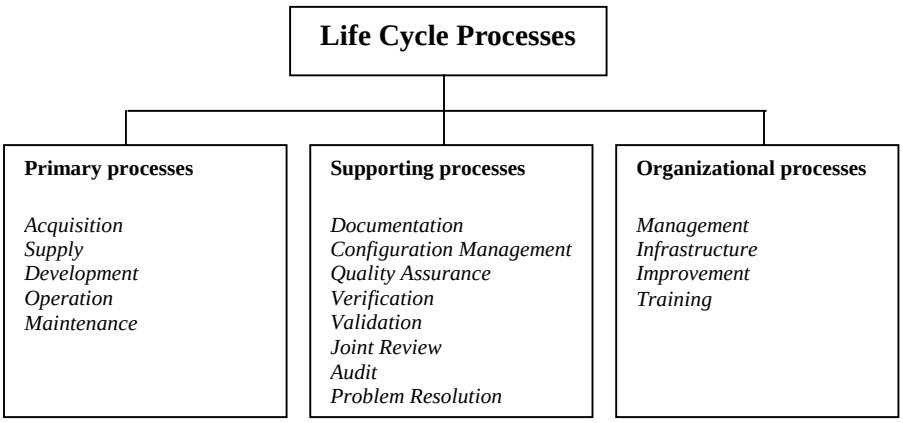

Figure 9: The ISO life cycle processes.

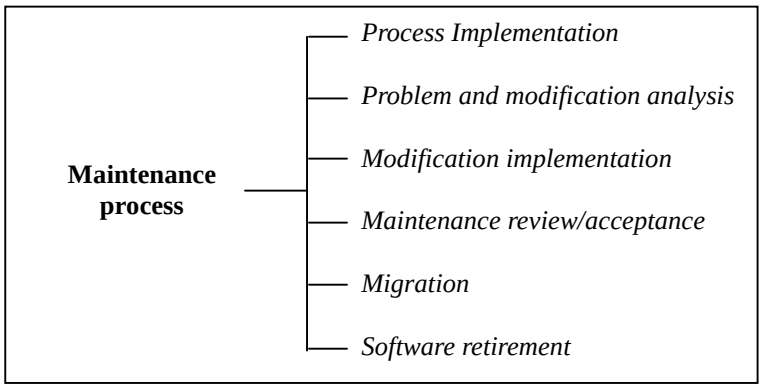

Figure 10: The ISO maintenance process.

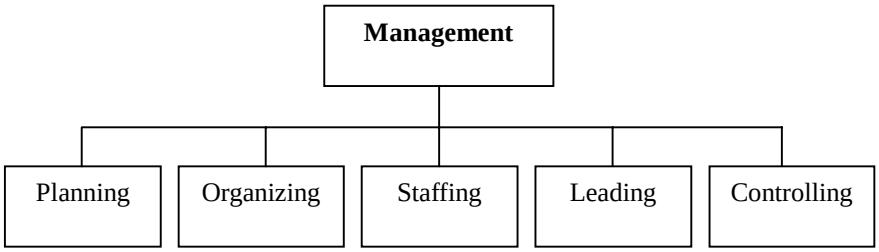

Figure 11: The functions of project management.

1. Introduction
Describes the purpose, goals, and scope of the software maintenance effort; determines deviations from the standard.

2. References
Identifies the documents that pose constraints on the maintenance effort and any other supporting documents.

3. Definitions
Defines or references all terms required to understand the plan.

4. Software Maintenance Overview
Describes organization, scheduling priorities, resources, responsibilities, tools, techniques, and methods used in the maintenance process.

- 4.1 Organization
- 4.2 Scheduling Priorities
- 4.3 Resource Summary
- 4.4 Responsibilities
- 4.5 Tools, Techniques, and Methods

5. Software Maintenance Process
Identifies the actions to perform for each phase of the maintenance process; actions are to be defined in terms of input, output, process, and control.

- 5.1 Problem/modification identification/classification and prioritization
- 5.2 Analysis
- 5.3 Design
- 5.4 Implementation
- 5.5 System Testing
- 5.6 Acceptance Testing
- 5.7 Delivery

6. Software Maintenance Reporting Requirements
Describes how information will be collected and provided to members of the maintenance organization.

7. Software Maintenance Administrative Requirements
Describes the standards, practices and rules for anomaly resolution and reporting.

- 7.1 Anomaly Resolution and Reporting
- 7.2 Deviation Policy
- 7.3 Control Procedures
- 7.4 Standards, Practices, and Conventions
- 7.5 Performance Tracking
- 7.6 Quality Control of Plan

8. Software Maintenance Documentation Requirements
Describes the procedures to be followed in recording and presenting the outputs of the maintenance process.

Figure 12: An example of a maintenance plan [41].

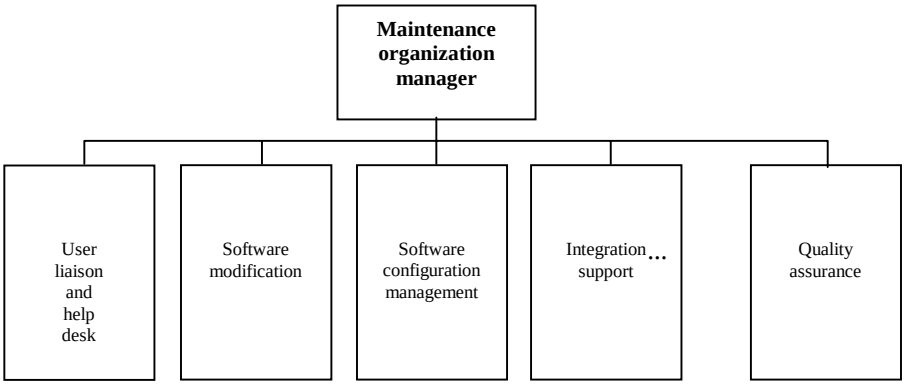

Figure 13: A functional organization.

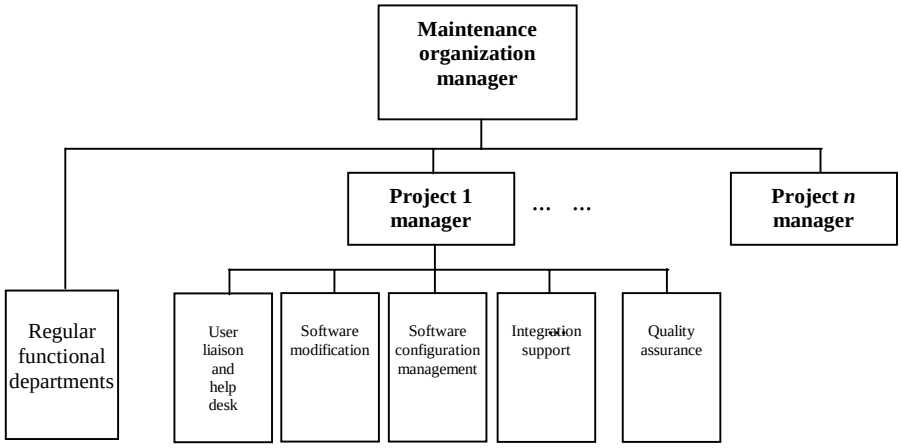

Figure 14: A project organization.

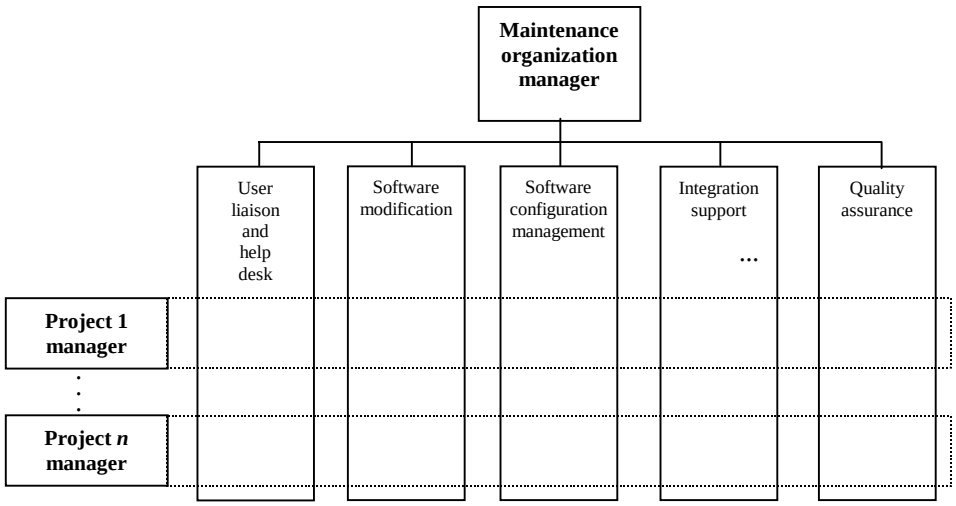

Figure 15: A matrix organization.

	Manager	
Decision	**Functional**	**Project**
Change the budget	Approves	Proposes
Commit resources	Approves	Proposes
Change requirements	Advises	Decides
Change release plan	Advises	Decides

Figure 16: An example of conflict resolution table.

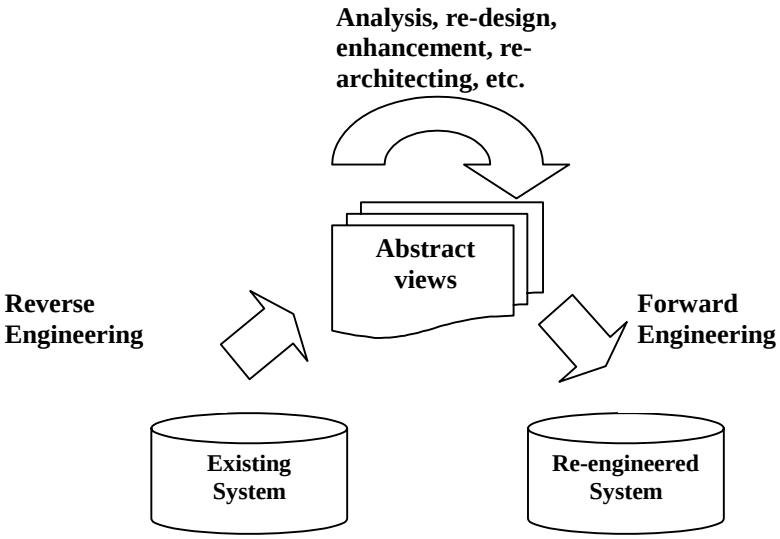

Figure 17: Reverse engineering and re-engineering.

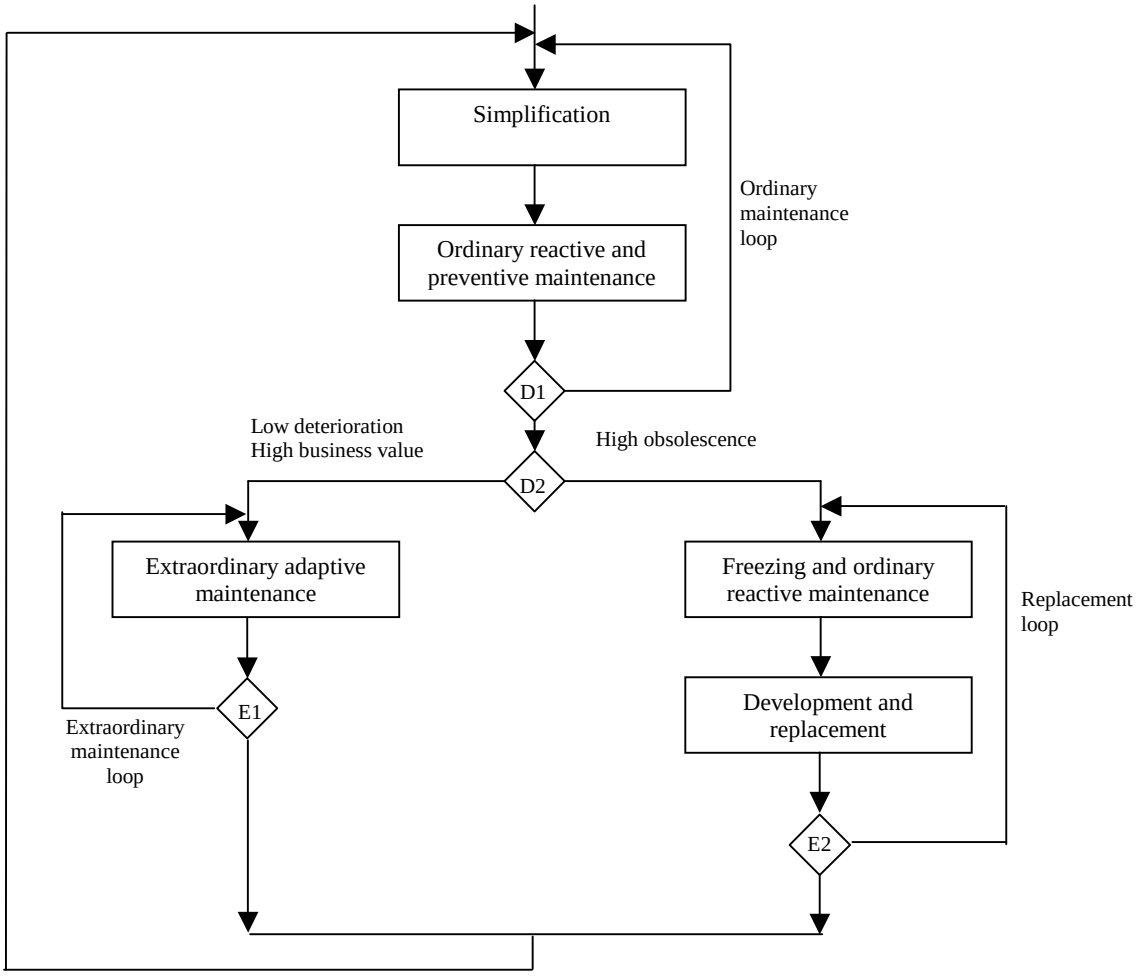

Figure 18: A life cycle model for legacy systems.