LESSON

9

SOFTWARE MAINTENANCE

CONTENTS

9.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Discuss categories of maintenance and problems during maintenance
- Explain maintenance process and maintenance models
- Describe reverse engineering and software re-engineering

9.1 INTRODUCTION

Software maintenance is concerned with modifying software once it is delivered to a customer. Another term used is "Software Evolution". Software maintenance is different than hardware maintenance. Software does not wear out. Software modifications include:

- Correcting errors in the software

- Adapting software to a new environment
- Enhancing functionality or improving performances

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

"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.

"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. It definitively dispels the image that software maintenance is all about fixing bugs or mistakes.

Impact of Software Maintenance

- Much of the software that is widely used is 10-15 years old.
- Much of it contains poorly organized design, sloppy and structured coding, confusing and cryptic logic, and little or no documentation.

9.2 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 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 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, 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: "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.

ISO 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 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 redefines the Lientz and Swanson categories of corrective, adaptive, and perfective maintenance, and adds preventive maintenance as a fourth category. The definitions are as follows:

- *Corrective Maintenance:* Changes to correct defects. Reactive modification of a software product performed after delivery to correct discovered faults.
- *Adaptive Maintenance:* Changes to adapt to new operating systems, hardware platforms, etc. Modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.
- *Perfective Maintenance:* Changes requested/needed for enhancement of the software. Modification of a software product performed after delivery to improve performance or maintainability.
- *Preventative Maintenance:* Changes made to software to prevent failures or improve its maintainability. Unscheduled corrective maintenance performed to keep a system operational."

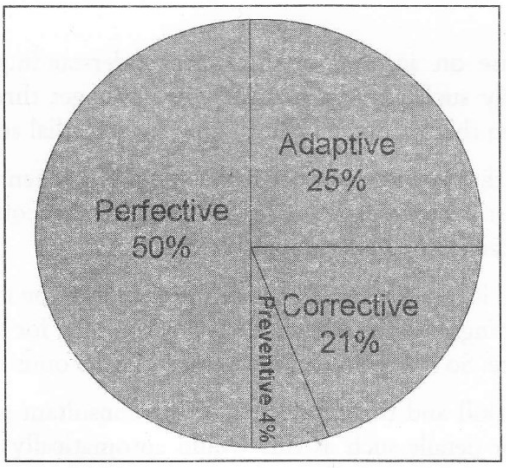

Figure 9.1: Use of Maintenance Time

Implementation is one phase of the SDLC, which is ridden with the maximum number of problems. The issues could range from the mundane to technical, from the predictable to the preposterous.

Handling these issues requires an enormous amount of managerial skill and the ability to foresee these problems.

Undoubtedly, expertise in this area can come only with experience. There is virtually no substitute for that. However here we shall attempt to look at some of the more recurrent problems - a compilation that has been arrived at from the experience drawn from various software projects and superimposed on our case study to see what could possibly go wrong during implementation.

The problem of a sudden increase in length of a field on account of a steep price hike cum a govt. order results in the field length been inadequate, the outcome of which leads to an unusable module. The scenario depicts a – conversation between department head and the invoicing clerk.

Departmental Head: This is a very important order for us. So, I do not want any slip-ups in processing it. Make sure all formalities are taken care of and that all documents are ready for dispatch of material.

Invoicing Clerk: Yes Sir, I will work on it immediately.

The invoicing clerk does all the necessary follow-up and when it finally comes to the generation of the invoice on the computer, he finds he is very badly stuck The program keeps giving an error message saying "field too large".

What is he to do?

Problem Analysis

Problems of this nature arise on account of improper understanding of the organization and its business needs. Unfortunately such gaps in analysis will never get thrown up till there is an actual exigency of this kind and then there is precious little time for remedial action.

The developers might argue that given the fact that the system has been extensively walkthrough with the user, it is actually the user's responsibility to point out such inadequacies. Since the user has given her approval, the developer cannot be held responsible.

While theoretically this stand is indeed correct, the reality is that in the majority of the cases, there will be no user who scrutinizes things like the data dictionary so closely; for the simple reason that she does not understand its importance. So it is pointless to assume that the onus lies only with the users.

This is where the technical skill and the experience of the consultant and the analyst count a lot. A number of apparently minor details such as this would automatically be given due attention by an experienced consultant.

For they know that understanding the business and its likely growth is an extremely important aspect during the development of any application.

9.3 PROBLEMS DURING MAINTENANCE

Many studies at the private, University and government level have been conducted to learn about maintenance requirements for information systems. These studies reveal the following facts:

(a) From 60 to 90 percent of the overall cost of software during the life of a system is spent on maintenance.

(b) Often maintenance is not done very efficiently.

(c) Software demand is growing at a faster rate than supply. Many programmers are spending more time on system maintenance than on new software development.

9.4 MAINTENANCE PROCESS

The process 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. There is a core set of activity that is 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 and the latter as a part of its standard on life cycle processes. The next two sections describe the maintenance processes defined by these two documents.

The IEEE standard organizes the maintenance process in seven phases. 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.

1. *Problem/modification identification, dassification, 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. This phase devises a preliminary plan for design, implementation, test, and delivery.

2. *Analysis:* Analysis is conducted at two levels: feasibility analysis and detailed analysis. Feasibility analysis identifies alternative solutions and assesses their impacts and costs, whereas detailed analysis defines the requirements for the modification, devises a test strategy, and develops an implementation plan.

3. *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.

4. *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.

5. *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.
6. *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.
7. *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.

ISO-12207 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. 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.

1. *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 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.
2. *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.
3. *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.
4. *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.
5. *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.

6. *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.

9.5 MAINTENANCE MODELS

A distinctive 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, which demonstrates the flow of changes from the old to the new version of the system. 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 propose 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. An example is iterative enhancement, which suggests structuring a problem to ease the design and implementation of successively larger/refined solutions. Iterative enhancement explains maintenance too. 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 efficient as the code changes. 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.

Maintenance is 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 suitable 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.

9.6 REVERSE ENGINEERING

The process of examine 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 is known as reverse engineering. 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 suitable to many problem areas. Six key objectives of reverse engineering: coping with complexity, generating alternate views, recovering lost information, detecting side effects, synthesizing higher abstractions, and facilitating reuse. Reverse engineering is a key that supports 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, finding objects in procedural programs, discovering architectures, deriving conceptual data models, detecting duplications, transforming binary programs into source code, renewing user interfaces, parallelizing sequential programs, and translating, downsizing, migrating], and wrapping legacy code. Reverse engineering principles have also been applied to business process re-engineering to create a model of an existing enterprise. 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.

Software Maintenance propose 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).

The need for a high-level organizational example 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.

9.7 SOFTWARE RE-ENGINEERING

The practice of re-engineering a software system to better appreciate and maintain it has long been accepted within the software maintenance community. We can define 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". It indicates renovation and reclamation as possible synonyms; renewal is another commonly used term. Arnold gives a more comprehensive definition as follows:

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

Re-engineering involves 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. Software re-engineering has established important for several reasons. There are seven main reasons that demonstrate the significance 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, downsizing , translating , reducing maintenance costs , improving quality, and migrating and re-engineering data. The standard IEEE-1219 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 or 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 test bed 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 test bed 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 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.

9.8 CONFIGURATION MANAGEMENT

Configuration Management (CM) is the comprehensive recording and updating of information that explains an enterprise's hardware and software. Such information typically includes the versions and updates that have been applied to installed software packages and the locations and network addresses of hardware devices. Special configuration management software is available. When a system needs hardware or software upgrade, a computer technician can accesses the configuration management program and database to see what is currently installed. The technician can then make a more informed decision about the upgrade needed.

An advantage of a configuration management application is that the complete collection of systems can be reviewed to make sure any changes made to one system do not adversely affect any of the other systems.

Configuration management is also used in software development, where it is called Unified Configuration Management. Using UCM, developers can keep track of the source code, documentation, problems, changes requested, and changes made.

When you construct computer software, change happens. And because it happens, you need to control it effectively. Software configuration management is a set of activities that are designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling changes that are imposed, and auditing and reporting on the changes that are made.

Software Configuration Management

The software being an logical production has certain characteristics not commonly associated with physical products. These can be summarized as under:

It can be effortlessly copied and reproduced. It can also be transferred and transported easily from one machine to other.

It is moderately easy to change and modify the software. Often the changes/modification carried out on a copied software are minor and the changes may not to noticeable easily. such modifications however after the output of the software.

When software is dispersed to various locations, it is very difficult to ensure identical versions everywhere.

There is often a need to adjust the software to rectify the errors noticed at some stage, for a particular situation encountered or to accommodate changes in the policy, procedures etc.

The need for changes in software have been sum up as the first law system engineering as under:

"No matter where you are in the SDLC, the system will change & desire to change in will persist". even though the changes in application software are inevitable, such changes are the source of confusion among the software developers. When changes are made they alter the configuration of the software. Keeping track of these changes is important.

Software Configuration Management (SCM) is the art of organizing software development process so that minimum confusion is caused among the developers.

Definition: "SCM is the of identifying organizing and controlling modifications to the software being built by the team of developers. The objective is to maximize productivity by minimizing mistakes" SCM is applied though the software Engineering process and includes:

1. Identification of change
2. Controlling the change
3. Ensuring that the change is properly implemented
4. Communicating change to others involved in it.

The SCM is different than software maintenance as maintenance in normally after the software is delivered whereas SCM undertaken throughout the process of software development upto delivery. However, the process of SCM is appropriate to software maintenance also.

Primary goal of software engineering is to develop the ease with which changes can be accommodated and decrease the amounts of efforts expanded when changes must be made.

The output of SE process are divided into three broad categories:

1. Application programmes
2. Documents that describe the computer programmes for (practitioners)
3. Data structure or Data Base software configuration includes all these items.

Changes: The changes become predictable during the software development for following reasons.

1. Customers/users want to modify requirements
2. Management wants to modify the project approach
3. Practioners want to modify programming approach

As the project advances in the development, everyone acquires more knowledge and would like to incorporate changes based on this knowledge.

Baseline: Baseline helps us to control changes without impediments.

Software Configuration Items

1. System specification
2. Software Project Plan
3. (a) Software requirement Specification
   (b) Executable or 'paper' prototype
4. Preliminary user Manual

Object description in O.O.Ps.

5. Source code listing
6. Test plan and procedure
7. Design specification
   (a) Data Design
   (b) Architectural Design
   (c) Module Design
   (d) Interface Design

Test cases and recorded results

8. Operation and Installation Manuals
9. Executable
10. Database Description
    (a) Scheme & file structure
    (b) Initial content
11. Built user Manual
12. Maintenance Documents
    (a) Software problem report
    (b) Maintenance requests
    (c) Engineering change orders
13. Standards and procedures for Software Engineering

Software tools i.e. version of editor, compilers and other CASE tools are frozen. so as to finalize the tools, compilers, etc. with which the development can be done.

The most frequently used approach to software configuration management is using the base line and decimal numbering system.

9.9 ESTIMATION OF MAINTENANCE COST

Maintenance cost can account for over 60% of the total software life-cycle cost. Half of the maintenance cost can be consumed in trying to understand the software .About 20% of maintenance costs are spent on correcting errors. Many entry-level software engineers start off in maintenance. 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, 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 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 is: *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%. 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.

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 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 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. 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.

Most problems that are associated with software maintenance can be traced to deficiencies of the software development process. Sneidewind 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 identifies complexity, conformity, changeability, and invisibility as four essential difficulties of software and Rajlich adds discontinuity to this list.

Check Your Progress

Fill in the blanks

1. Analysis is conducted at two levels .....analysis and detailed analysis.
2. Evolutionary life cycle models suggest an .....approach to software maintenance.
3. ..... management is the detailed recording and updating of information that describes an enterprise's hardware and software.
4. Software maintenance is the process of .....a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.

9.10 LET US SUM UP

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. Maintenance can be categorized into corrective, adaptive, and perfective maintenance, preventive maintenance. IEEE and ISO have both addressed software maintenance, the first with a specific standard and the latter as a part of its standard on life cycle processes.

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 is known as process engineering.

9.11 KEYWORDS

Software Maintenance: The process of modifying a software system

Preventative Maintenance: Changes made to software to prevent failures

Process Implementation: This activity includes the tasks for developing plans and procedures for software maintenance

Software Re-engineering: The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form

9.12 QUESTIONS FOR DISCUSSION

1. Give at least two definitions of software engineering.
2. What are the various categories if software maintenance in IEEE and ISO?

3. What are the problems during maintenance?
4. Discuss reverse engineering and software re-engineering.
5. Explain the various maintenance processes.

Check Your Progress: Model Answers

1. Feasibility
2. Alternative
3. Configuration
4. Modifying

9.13 SUGGESTED READINGS

R.S. Pressman, *Software Engineering-A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Rajib Mall, *Fundamentals of Software Engineering*, PHI, 2nd Edition.

Sommerville, *Software Engineering*, Pearson Education, 6th Edition.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

LESSON

10

DOCUMENTATION

CONTENTS

10.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Describe the concept of documentation and classification of documentation
- Define user documentation
- Explain system documentation

10.1 INTRODUCTION

All large software development projects, irrespective of application, produce a large amount of associated documentation. For moderately sized systems, the documentation will probably fill several filing cabinets; for large systems, it may fill several rooms. A high proportion of software process costs are incurred in producing this documentation. Furthermore, documentation errors and omissions can lead to errors by end-users and consequent system failures with their associated costs and disruption. Therefore, managers and software engineers should pay as much attention to documentation and its

associated costs as to the development of the software itself. The documents associated with a software project and the system being developed has a number of associated requirements:

- They should act as a communication medium between members of the development team.
- They should be a system information repository to be used by maintenance engineers.
- They should provide information for management to help them plan, budget and schedule the software development process.
- Some of the documents should tell users how to use and administer the system.

Fulfilling these requirements requires diverse types of document from informal working documents through to professionally produced user manuals. Software engineers are usually responsible for producing most of this documentation although professional technical writers may assist with the final polishing of externally released information. The documentation which may be produced during the software process, to give some hints on ways of writing effective documents and to describe processes involved in producing these documents.

10.2 PROCESS AND PRODUCT DOCUMENTATION

For large software projects, it is usually the case that documentation creates being generated well before the development process begins. A proposal to develop the system may be produced in response to a request for tenders by an external client or in response to other business strategy documents. For some types of system, a comprehensive requirements document may be produced which defines the features required and expected behavior of the system. During the development process itself, all sorts of different documents may be produced ñ project plans, design specifications, test plans etc. It is not possible to define a specific document set that is required ñ this depends on the contract with the client for the system, the type of system being developed and its expected lifetime, the culture and size of the company developing the system and the development schedule that it expected. However, we can generally say that the documentation produced falls into two classes:

1. ***Process documentation:*** These documents record the process of development and maintenance. Plans, schedules, process quality documents and organizational and project standards are called as process documentation.
2. ***Product documentation:*** This documentation enlightens the product that is being developed. System documentation describes the product from the point of view of the engineers developing and maintaining the system; user documentation provides a product description that is oriented towards system users.

Process documentation is formed so that the development of the system can be managed. Product documentation is used after the system is operational but is also essential for management of the system development. The creation of a document, such as a system specification, may represent an important milestone in the software development process.

10.2.1 Process Documentation

Successful management requires the process being managed to be visible. Because software is intangible and the software process involves apparently similar cognitive tasks rather than obviously different

physical tasks, the only way this visibility can be achieved is through the use of process documentation. Process documentation falls into a number of categories:

- *Plans, estimates and schedules:* These are documents created by managers which are used to predict and to control the software process.
- *Reports:* These documents report how resources were used during the process of development.
- *Standards:* These are documents which set out how the process is to be implemented. These may be developed from organizational, national or international standards.
- *Working papers:* These are often the principal technical communication documents in a project. They record the ideas and thoughts of the engineers working on the project, are interim versions of product documentation, describe implementation strategies and set out problems which have been identified. They often, implicitly, record the rationale for design decisions.
- *Memos and electronic mail messages:* These record the details of everyday communications between managers and development engineers.

The major attribute of process documentation is that most of it becomes outdated. Plans may be drawn up on a weekly, fortnightly or monthly basis. Progress will normally be reported weekly. Memos record thoughts, ideas and intentions which change. Although the interest of software historians, much of this process information is of little real use after it has gone out of date and there is not normally a need to preserve it after the system has been delivered. However, there are some process documents that can be useful as the software evolves in response to new requirements. For example, test schedules are of value during software evolution as they act as a basis for re-planning the validation of system changes. Working papers which explain the reasons behind design decisions (design rationale) are also potentially valuable as they discuss design options and choices made. Access to this information helps avoid making changes which conflict with these original decisions. Ideally, of course, the design rationale should be extracted from the working papers and separately maintained. Unfortunately this hardly ever happens.

10.2.2 Product Documentation

Product documentation explains the delivered software product. Unlike most process documentation, it has a relatively long life. It must evolve in step with the product which it describes. Product documentation includes user documentation which tells users how to use the software product and system documentation which is principally intended for maintenance engineers.

User Documentation

Users of a system are not all the same. The creator of documentation must structure it to cater for different user tasks and different levels of knowledge and experience. It is particularly important to differentiate between end-users and system administrators:

- *End-users* use the software to support some task. This may be flying an aircraft, managing insurance policies, writing a book, etc. They want to know how the software can help them. They are not interested in computer or administration details.
- *System administrators* are responsible for managing the software used by end-users. This may entail acting as an operator if the system is a large mainframe system, as a network manager is the

system involves a network of workstations or as a technical guru who fixes end-users software problems and who liaises between users and the software supplier.

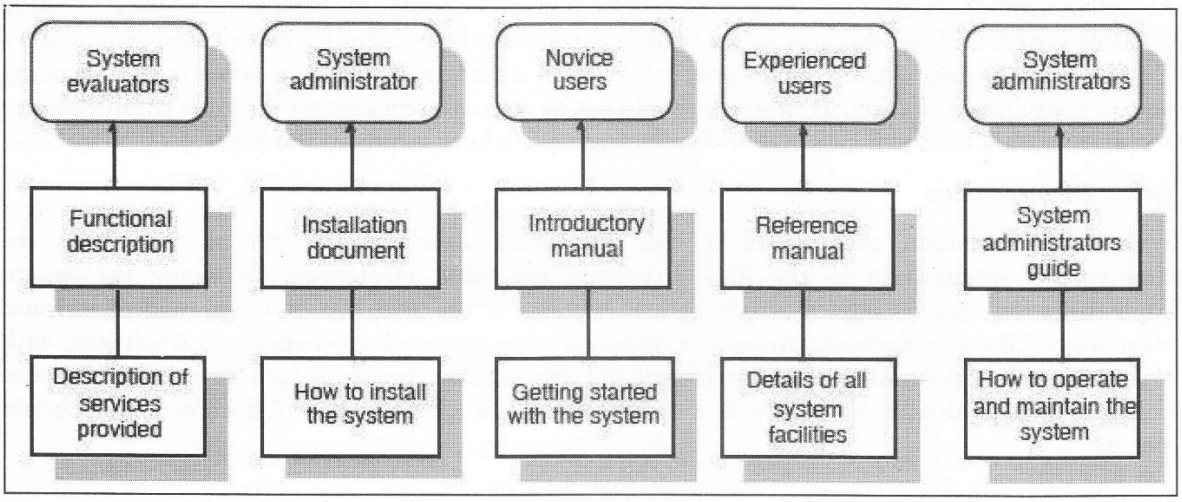

Figure 10.1: Different Types of user Documentation

To provide these different classes of user and different levels of user expertise, there are at least 5 documents (or perhaps chapters in a single document) which should be delivered with the software system (Figure10.1).

The *functional description* of the system outlines the system requirements and briefly describes the services provided. This document should provide an impression of the system. Users should be able to read this document with an introductory manual and decide if the system is what they need.

The *system installation document* is intended for system administrators. It should provide details of how to install the system in a particular environment. It should hold a description of the files making up the system and the minimal hardware configuration required. The permanent files which must be established, how to start the system and the configuration dependent files which must be changed to tailor the system to a particular host system should also be described. The use of automated installers for PC software has meant that some suppliers see this document as unnecessary. In fact, it is still required to help system managers discover and fix problems with the installation.

The *introductory manual* should present an informal introduction to the system, describing its normal usage. It should explain how to get started and how end-users might make use of the common system facilities. It should be liberally illustrated with examples. Inevitably beginners, whatever their background and experience, will make mistakes. Easily discovered information on how to recover from these mistakes and restart useful work should be an integral part of this document.

The *system reference manual* should explain the system facilities and their usage, should provide a complete listing of error messages and should describe how to recover from detected errors. It should be complete. Formal descriptive techniques may be used. The style of the reference manual should not be unnecessarily pedantic and turgid, but completeness is more important than readability.

A more general *system administrator's guide* should be provided for some types of system such as command and control systems. This should explain the messages generated when the system interacts with other systems and how to react to these messages. If system hardware is involved, it might also

explain the operator's task in maintaining that hardware. For example, it might describe how to clear faults in the system console, how to connect new peripherals, etc.

As well as manuals, other, easy-to-use documentation might be provided. A rapid reference card listing available system facilities and how to use them is particularly convenient for experienced system users. On-line help systems, which contain brief information about the system, can save the user spending time in consultation of manuals although should not be seen as a replacement for more comprehensive documentation. I briefly discuss on-line documentation in a later section in this paper.

System Documentation

System documentation contains all of the documents describing the system itself from the requirements specification to the final acceptance test plan. Documents describing the design, implementation and testing of a system are essential if the program is to be understood and maintained. Like user documentation, it is important that system documentation is structured, with overviews leading the reader into more formal and detailed descriptions of each aspect of the system. For large systems that are developed to a customer's specification, the system documentation should include:

- The requirements document and an associated rationale.
- A document describing the system architecture.
- For each program in the system, a description of the architecture of that program.
- For each component in the system, a description of its functionality and interfaces.
- Program source code listings. These should be commented where the comments should explain complex sections of code and provide a rationale for the coding method used. If meaningful names are used and a good, structured programming style is used, much of the code should be self-documenting without the need for additional comments. This information is now normally maintained electronically rather than on paper with selected information printed on demand from readers.
- Validation documents describing how each program is validated and how the validation information relates to the requirements.
- A system maintenance guide who describes known problems with the system describes which parts of the system are hardware and software dependent and which describes how evolution of the system has been taken into account in its design.

An ordinary system maintenance problem is ensuring that all representations are kept in step when the system is changed. For smaller systems and systems that are developed as software products, system documentation is usually less comprehensive. This is not necessarily a good thing but schedule pressures on developers mean that documents are simply never written or, if written, are not kept up to date. These pressures are sometimes inevitable but, in my view, at the very least you should always try to maintain a specification of the system, an architectural design document and the program source code.

Unfortunately, documentation maintenance is often ignored. Documentation may become out of step with its associated software, causing problems for both users and maintainers of the system. The natural tendency is to meet a deadline by modifying code with the intention of modifying other documents later.

A lot, pressure of work means that this modification is continually set aside until finding what is to be changed becomes very difficult indeed. The best solution to this problem is to support document maintenance with software tools which record document relationships, remind software engineers when changes to one document affect another and record possible inconsistencies in the documentation.

10.3 CLASSIFICATION SCHEMES

10.3.1 Document Quality

Unfortunately, much computer system documentation is poorly written, difficult to understand, out-of-date or incomplete. Although the situation is improving, many organizations still do not pay enough attention to producing system documents which are well-written pieces of technical prose. Document quality is as significant as program quality. Without information on how to use a system or how to understand it, the utility of that system is degraded. Attaining document quality requires management commitment to document design, standards, and quality assurance processes. Producing good documents is neither easy nor cheap and many software engineers find it more difficult that producing good quality programs.

10.3.2 Document Structure

Document structure has a major impact on readability and usability The document structure is the way in which the material in the document is organized into chapters and, within these chapters, into sections and sub- sections and it is important to design this carefully when creating documentation. As with software systems, you should design document structures so that the dissimilar parts are as independent as possible. This allows each part to be read as a single item and reduces problems of cross-referencing when changes have to be made. Structuring a document properly also allows readers to locate information more easily. As well as document components such as contents lists and indexes, well-structured documents can be skim read so that readers can quickly locate sections or sub-sections that are of most attention to them.

10.3.3 Documentation Standards

Documentation standards act as a basis for document quality assurance. Documents produced according to appropriate standards have a reliable appearance, structure and quality. I have already introduced the IEEE standard for user documentation in the previous section and will discuss this standard in more detail shortly. However, it is not only standards that focus on documentation that are pertinent. Other standards that may be used in the documentation process are:

1. *Process standards:* Process standards define the process which should be followed for high-quality document production.
2. *Product standards:* These are standards which administer the documents themselves.
3. *Interchange standards:* It is increasingly important to exchange copies of documents via electronic mail and to store documents in databases. Interchange standards make sure that all electronic copies of documents are compatible.

Table 10.1: Suggested Components in a Software user Document

Component	Description
Identification data	Data such as a title and identifier that uniquely identifies the document.
Table of contents	Chapter/section names and page numbers.
List of illustrations	Figure numbers and titles
Introduction	Defines the purpose of the document and a brief summary of the contents
Information for use of the documentation	Suggestions for different readers on how to use the documentation effectively.
Concept of operations	An explanation of the conceptual background to the use of the software.
Procedures	Directions on how to use the software to complete the tasks that it is designed to support.
Information on software commands	A description of each of the commands supported by the software.
Error messages and problem resolution	A description of the errors that can be reported and how to recover from these errors.
Glossary	Definitions of specialized terms used.
Related information sources	References or links to other documents that provide additional information
Navigational features	Features that allow readers to find their current location and move around the document.
Index	A list of key terms and the pages where these terms are referenced.
Search capability	In electronic documentation, a way of finding specific terms in the document.

Check Your Progress

State whether the following are true or false:

1. Process documentation is produced so that the development of the system can be managed.
2. System documentation does not include all of the documents describing the system itself from the requirements specification to the final acceptance test plan.
3. The system installation document is intended for system administrators.

10.4 LET US SUM UP

The lesson discusses the many components like documentation, user documentation, system documentation and classification schemes. The more people learned about these components the more effective they will become in their software documentation. All large software development projects, irrespective of application, generate a large amount of associated documentation.

10.5 KEYWORDS

Process Documentation: These documents records the process of development and maintenance.

Product Documentation: This documentation describes the product that is being developed.

User Documentation: Users of a system are not all the same. The producer of documentation must structure it to cater for different user tasks and different levels of expertise and experience.

System Documentation: System documentation includes all of the documents describing the system itself from the requirements specification to the final acceptance test plan.

10.6 QUESTIONS FOR DISCUSSION

1. What are the associative requirements of the documents?
2. Write a short note on user documentation and system documentation.
3. Discuss the document standards.

Check Your Progress: Model Answers

1. True
2. False
3. True

10.7 SUGGESTED READINGS

R.S. Pressman, *Software Engineering - A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Rajib Mall, *Fundamentals of Software Engineering*, PHI, 2nd Edition.

Sommerville, *Software Engineering*, Pearson Education, 6th Edition.

Richard Fairpy, *Software Engineering* Concepts, Tata McGraw Hill, 1997.