Course Name: M. Tech
Semester: 2nd
Paper Name: Object Oriented Software Engineering (MTCS-203B)

Topic: Need for Maintenance.

Need for Maintenance

Introduction:

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Therefore it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

Categories of Maintenance:

There are basically three types of software maintenance. These are:

• **Corrective**: Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.

• **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

• **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Problems associated with software maintenance

Software maintenance work typically is much more expensive than what it should be and takes more time than required. In software organizations, maintenance work is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities. Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

Software reverse engineering

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing of its functionalities. . A program can be reformatted using any of the several available pretty printer programs which layout the program neatly. Many legacy software products with complex control structure and unthoughtful variable names are difficult to comprehend. Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible.

Complex Estimation of approximate maintenance cost

It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT).

Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

ACT=KLOC (ADDED)+KLOC(DELETED)/KLOC(TOTAL)

CASE tool and its scope

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool means any tool used to automate some activity associated with software development. Many CASE tools are available. Some of these CASE tools assist in phase related tasks such as specification, structured analysis, design, coding, testing, etc.; and others to non-phase activities such as project management and configuration management.

CASE environment

Although individual CASE tools are useful, the true power of a tool set can be realized only when these set of tools are integrated into a common framework or environment. CASE tools are characterized by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software development artifacts.

Benefits of CASE

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Some of those benefits are:

• A key benefit arising out of the use of a CASE environment is cost saving through all development phases. Different studies carry out to measure the impact of CASE put the effort reduction between 30% to 40%. Use of CASE tools leads to considerable improvements to quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development and the chances of human error are considerably reduced.

• CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced and therefore chances of inconsistent documentation is reduced to a great extent.

• CASE tools take out most of the drudgery in a software engineer's work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.

• CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.

• Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach. Structured analysis and design with CASE tools several diagramming techniques are used for structured analysis and structured design. The following supports might be available from CASE tools.

• A CASE tool should support one or more of the structured analysis and design techniques.

• It should support effortlessly drawing analysis and design diagrams.

• It should support drawing for fairly complex diagrams, preferably through a hierarchy of levels.

• The CASE tool should provide easy navigation through the different levels and through the design and analysis.

• The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Whenever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there arises heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

Organic, Semidetached and Embedded software projects

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Organic: A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached: A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded: A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following

Expressions:

Effort = a1 x (KLOC)POWa 2 PM

Tdev = b1 x (Effort)POWb 2 Months

Where:

• KLOC is the estimated size of the software product expressed in Kilo Lines of Code,

• a1, a2, b1, b2 are constants for each category of software products,

• Tdev is the estimated time to develop the software, expressed in months,

• Effort is the total effort required to develop the software product, expressed in person months (PMs).

Estimation of development effort for the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : Effort = 2.4(KLOC) POW1.05 PM

Semi-detached: Effort = 3.0(KLOC) POW1.12 PM

Embedded : Effort = 3.6(KLOC) POW1.20 PM

Estimation of development time for the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : Tdev = 2.5(Effort) POW0.38 Months

Semi-detached: Tdev = 2.5(Effort) POW0.35 Months

Embedded : Tdev = 2.5(Effort) POW 0.32 Months

Example:

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = 2.4 x (32)

1.05

= 91 PM

Nominal development time = 2.5 x (91)

0.38

= 14 months

Cost required to develop the product = 14 x 15,000

= Rs. 210,000/

Intermediate COCOMO model

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three.

Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.

Development Environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

Complete COCOMO model

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems may be different, but also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

Software Risk Analysis and Management

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague of software project. A risk is a potential problem; it might happen, it might not. But regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

What is Risk?

Risk is defined as "The possibility of suffering harm or loss; danger." Even if we're not familiar with the formal definition, most of us have an innate sense of risk. We are aware of the potential dangers that permeate even simple daily activities, from getting injured when crossing the street to having a heart attack because our cholesterol level is too high. Although we prefer not to dwell on the myriad of hazards that surround us, these risks shape many of our behaviors. Experience (or a parent) has taught us to look both ways before stepping off the curb and most of us at least think twice before ordering a steak. Indeed, we manage personal risks every day.

Risks in Software Project Management

Unlike the hazards of daily living, the dangers in the young and emerging field of software engineering must often be learned without the benefit of lifelong exposure. A more deliberate approach is required. Such an approach involves studying the experiences of successful project managers as well as keeping up with the leading writers and thinkers in the field. One such writer in the area of risk is Dr. Barry W. Boehm. In his article "Software Risk Management: Principles and Practices" he lists the following top 10 software risk items:

1. Personnel Shortfalls
2. Unrealistic schedules and budgets
3. Developing the wrong functions and properties
4. Developing the wrong user interface
5. Gold-plating
6. Continuing stream of requirements changes
7. Shortfalls in externally furnished components
8. Shortfalls in externally performed tasks
9. Real-time performance shortfalls

10. Straining computer-science capabilities

How to Manage

In the same article, Dr. Boehm describes risk management as being comprised of the following activities:

- ➢ Risk Assessment (figuring out what the risks are and what to focus on)
  - ✓ - making a list of all of the potential dangers that will affect the project
  - ✓ - assessing the probability of occurrence and potential loss of each item listed
  - ✓ - ranking the items (from most to least dangerous)
- ➢ Risk Control (doing something about them)
  - ✓ - coming up with techniques and strategies to mitigate the highest ordered risks
  - ✓ - implementing the strategies to resolve the high order risks factors
  - ✓ - monitoring the effectiveness of the strategies and the changing levels of risk throughout the project