**UNIT-V**

**CHAPTER-II**

**SOFTWARE MAINTENANCE**

Software maintenance denotes any changes made to a software product after it has been delivered to the customer.

Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use. For example, a car tyre wears out due to use.

On the other hand, software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platforms, etc.

**CHARACTERISTICS OF SOFTWARE MAINTENANCE**

Software maintenance is becoming an important activity of a large number of organisations. 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 changes, 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 or the software needs to run over hand held devices. Thus, every software product continues to evolve after its development through maintenance efforts.

**Types of Software Maintenance**

There are three types of software maintenance, which are described as follows:

**Corrective:** Corrective maintenance of a software product is necessary to overcome the failures 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 any new features that the users may want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

**Characteristics of Software Evolution**

Lehman and Belady studied the characteristics of evolution of several software products [1980]. They expressed their observations in the form of laws. Their important laws are presented in the following subsection. But a word of caution here is that these are generalisations and may not be applicable to specific cases. Further, most of their observations concern large software projects and may not be appropriate for the maintenance and evolution of very small products.

**Lehman’s first law:**

A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs and therefore tend to incur higher maintenance efforts.

This law clearly shows that every product irrespective of how well designed must undergo maintenance.

In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need maintenance. In fact, good products are maintained and bad products are thrown away.

**Lehman’s second law:**

The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that usually maintenance activities result in patch work. It is rarely the case that members of the original development team are part of the maintenance team.

The maintenance team, therefore, often has a partial and inadequate understanding of the architecture, design, and code of the software. Therefore, any modifications tend to be ugly and more complex than they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

**Lehman’s third law:**

Over a program’s lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore, this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

**Special Problems Associated with Software Maintenance**

Software maintenance work currently is typically much more expensive than what it should be and takes more time than required. The reasons for this situation are the following:

Software maintenance work in organisations 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 organisation 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. Though the word legacy implies “aged” software, but there is no agreement on what exactly is a legacy system. It is prudent to define a legacy system as any software system that is hard to maintain.

The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product.

Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

**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 any of its functionalities. A way to carry out these cosmetic changes is shown schematically in Figure 13.1.

A program can be reformatted using any of the several available Pretty Printer programs which layout the program neatly. Many legacy software products are diﬃcult to comprehend with complex control structure and unthoughtful variable names. Assigning meaningful variable names is important that meaningful variable names is the most helpful code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible.

Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.



After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.



**SOFTWARE MAINTENANCE PROCESS MODELS**

Before discussing process models for software maintenance, we need to analyse various activities involved in a typical software maintenance project. The activities involved in a software maintenance project are not unique and depend on several factors such as:

1. the extent of modification to the product required,
2. (ii) the resources available to the maintenance team,
3. (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.), (iii) the expected project risks, etc.

 When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents.

However, more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Since the scope (activities required) for different maintenance projects vary widely, no single maintenance process model can be developed to suit every kind of maintenance project. However, two broad categories of process models can be proposed

**First model**

The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in Figure 13.3. In this approach, the project starts by gathering the requirements for changes.

The requirements are next analysed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code.

The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the re-engineered system becomes easier as the program traces of both the systems can be compared to localise the bugs.



**Second model**

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as *software re-engineering*. This process model is depicted in Figure 13.4.

The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analysed (abstracted) to extract the module specifications. The module specifications are then analysed to produce the design. The design is analysed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification.

At this point a forward engineering is carried out to produce the new code. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased eﬃciency. The eﬃciency improvements are brought about by a more eﬃcient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15 per cent (see Figure 13.5).

 

Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2 as follows:

 Re-engineering might be preferable for products which exhibit a high failure rate.

 Re-engineering might also be preferable for legacy products having poor design and code structure.



**ESTIMATION OF MAINTENANCE COST**

We had earlier pointed out that maintenance efforts require about 60 per cent 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.



where, KLOC added is the total kilo lines of source code added during maintenance. KLOC deleted is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and code deleted.

The annual change traﬃc (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

Maintenance cost = ACT *×* Development cost

Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.