SOFTWARE ENGINEERING
UNIT-6

1. **Software Maintenance:** Software maintenance,
2. Maintenance Process Models,
3. Maintenance Cost,
4. Software Configuration Management.
5. **Software Reuse:** what can be Reused?
6. Why almost No Reuse So Far?
7. Basic Issues in Reuse Approach,
8. Reuse at Organization Level.

1. SOFTWARE MAINTENANCE

- The mention of the word maintenance brings up the image of a screw driver, wielding mechanic with soiled hands holding onto a bagful of spare parts.
- It would be the objective of this chapter to give idea of the software maintenance projects, and to familiarise you with the latest techniques in software maintenance.
- Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is necessity for almost any kind of product.
- However, most products need maintenance due to the wear and tear caused by 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

- In this section, we first classify the different maintenance efforts into a few classes.

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 either 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.

Characteristics of Software Evolution

- Lehman and Belady have studied the characteristics of evolution of several software products [1980].
- They have 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 and also most of these 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.
- **Lehman's second law:** The structure of a program tends to degrade as more and more maintenance is carried out on it.
- **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.

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.

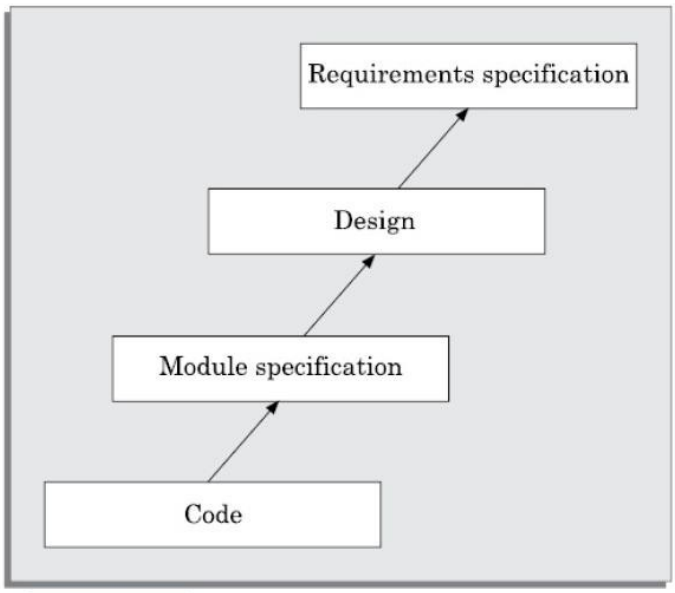


Figure 13.1: A process model for reverse engineering.

- A way to carry out these cosmetic changes is shown schematically in Figure 13.1.
- After the cosmetic changes have been carried out on a legacy software, the proces 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.

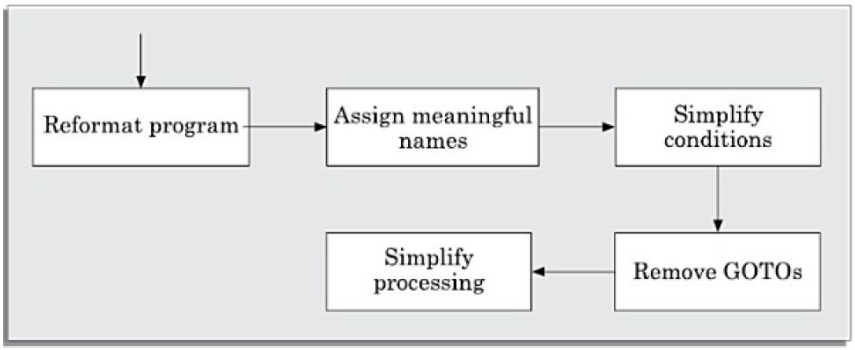


Figure 13.2: Cosmetic changes carried out before reverse engineering.

2. SOFTWARE MAINTENANCE PROCESS MODELS

- The activities involved in a software maintenance project are
  - (i) the extent of modification to the product required,
  - (ii) the resources available to the maintenance team,
  - (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.),
  - (iv) 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.
- 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.

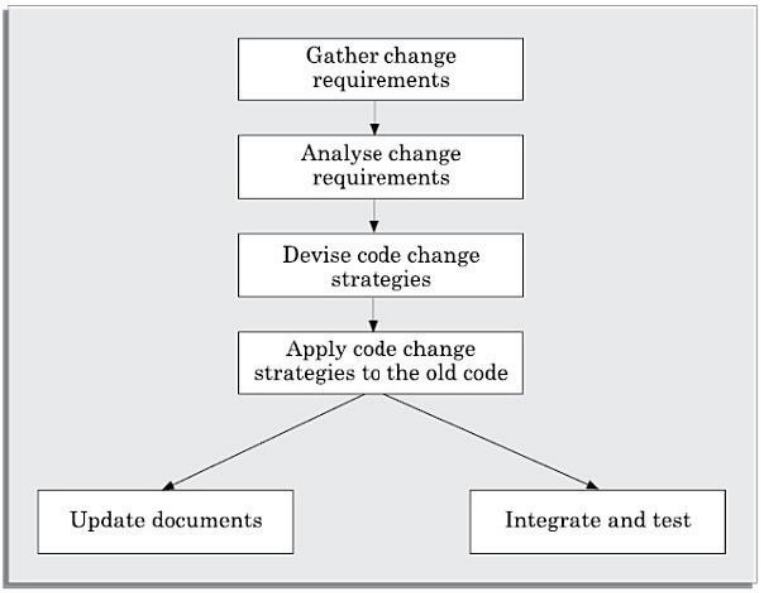


Figure 13.3: Maintenance process model 1.

- 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 reengineered 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.

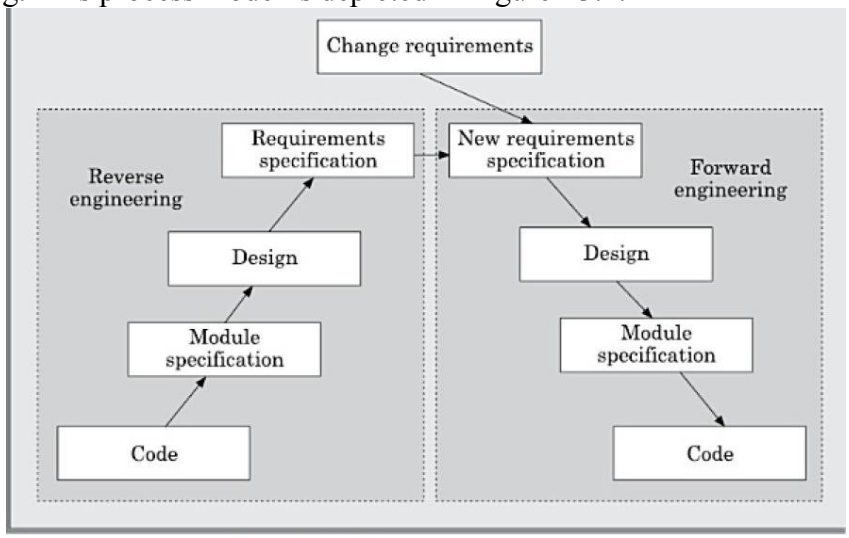


Figure 13.4: Maintenance process model 2.

- 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.

3. 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 **Co**nstructive **Co**st **Mo**del - COCOMO cost estimation model.
- Boehm's maintenance cost estimation is made in terms of a quantity called the **A**nnual **C**hange **T**raffic (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.

4. SOFTWARE CONFIGURATION MANAGEMENT

- The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g., source code, design document, SRS document, test document, user's manual, etc.
- These objects are usually referred to and modified by a number of software developers through out the life cycle of the software.
- The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.
- The configuration of the software is the state of all project deliverables at any point of time; and software configuration management deals with effectively tracking and controlling the configuration of a software during its life cycle.

Software revision versus version

- A new version of software is created when there is significant change in functionality, technology, or the hardware it runs on, etc.
- On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc.
- Even the initial delivery might consist of several versions and more versions might be added later on.
- For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows and so on.
- As a software is released and used by the customer, errors are discovered that need correction.
- Enhancements to the functionalities of the software may also be needed.
- A new release of software is an improved system intended to replace an old one.

Necessity of Software Configuration Management

- There are several reasons for putting an object under configuration management. The following are some of the important problems.
- **Problems associated with concurrent access:** Possibly the most important reason for configuration management is to control the access to the different deliverable objects.
- **Providing a stable development environment:** When a project work is underway, the team members need a stable environment to make progress.
- **System accounting and maintaining status information:** System accounting denotes keeping track of who made a particular change to an object and when the change was made.
- **Handling variants:** Existence of variants (software versions) of a software product causes some peculiar problems. Suppose you have several variants of the same module, and find that a bug exists in one of them. Then, it has to be fixed in all versions and revisions.
- To do it efficiently, you should not have to fix it in each and every version and revision of the software separately. Making a change to one program should be reflected appropriately in all relevant versions and revisions.

Configuration Management Activities

- Configuration management is carried out through two principal activities:
- **Configuration identification:** It involves deciding which parts of the system should be kept track.
- **Configuration control:** Configuration control is the process of managing changes to controlled objects. It ensures that changes to a system happen smoothly. A *configuration management tool* helps to keep track the current state of the project. The configuration management tool enables the developer to change various components in a controlled manner.
- **Source code control system (SCCS) and RCS:** SCCS and RCS are two popular **configuration management tools** available on most Unix systems.

5. SOFTWARE REUSE

- Software products are expensive. Therefore, software project managers are always worried about the high cost of software development .
- A possible way to reduce development cost is to reuse parts from previously developed software.
- In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality.
- A reuse approach that is of late gaining prominence is component-based development. Component-based software development is different from the traditional software development in the sense that software is developed by assembling software from off-the-shelf components.
- Software development with reuse is very similar to a modern hardware engineer building an electronic circuit by using standard types of ICs and other components.

WHAT CAN BE REUSED?

- Before discussing the details of reuse techniques, it is important to deliberate about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:

  Requirements specification
  Design
  Code
  Test cases
  Knowledge

- Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts, reuse of knowledge occurs automatically without any conscious effort in this direction.
- A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

6. WHY ALMOST NO REUSE SO FAR?

- Engineers working in software development organisations often have a feeling that the current system that they are developing **is similar to the last few systems built**.

- However, no attention is paid on how not to duplicate what can be reused from previously developed systems.
- Everything is being built from the old system.
- The current system falls behind schedule and no one has time to figure out how the similarity between the current system and the systems developed in the past can be exploited.
- Even those organisations which start the process of reusing programs
- Creation of components that are reusable in different applications is a difficult problem.
- It is very difficult to anticipate the exact components that can be reused across different applications.
- But, even when the reusable components are carefully created and made available for reuse, programmers prefer to create their own, because the available components are difficult to understand and adapt to the new applications.
- In this context, the following observation is significant:
    - The routines of mathematical libraries are being reused very successfully by almost every programmer.
    - No one in their mind would think of writing a routine to compute sine or cosine.
    - Let us investigate why reuse of commonly used mathematical functions is so easy.
    - Everyone has clear ideas about what kind of argument should implement, the type of processing to be carried out and the results returned.
    - Secondly, mathematical libraries have a small interface.

7. BASIC ISSUES IN ANY REUSE PROGRAM

- The following are some of the basic issues that must be clearly understood for starting any reuse program:
  Component creation.
  Component indexing and storing.
  Component search.
  Component understanding.
  Component adaptation.
  Repository maintenance.

Component creation:

- For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important.

Component indexing and storing

- Indexing requires classification of the reusable components so that they can be easily searched when we look for a component for reuse.
- The components need to be stored in a *relational database management system* (RDBMS) or an *object-oriented database system* (ODBMS) for efficient access when the number of components becomes large.

Component searching

- The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

Component understanding

- The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component.
- To facilitate understanding, the components should be well documented and should do something simple.

Component adaptation

- Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand.
- However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

Repository maintenance

- A component repository once is created requires continuous maintenance.
- New components, as and when created have to be entered into the repository.
- The faulty components have to be tracked.
- Further, when new applications emerge, the older applications become . In this case, the obsolete components might have to be removed from the repository.

A REUSE APPROACH

- A promising approach that is being adopted by many organisations is to introduce a building block approach into the software development process. For this, the reusable components need to be identified after every development project is completed.
- The reusability of the identified components has to be enhanced and these have to be cataloged into a component library.
- It must be clearly understood that an issue crucial to every reuse effort is the identification of reusable components.
- Domain analysis is a promising approach to identify reusable components.
- The domain analysis approach to create reusable components.

Domain Analysis

- The aim of domain analysis is to identify the reusable components for a problem domain.

Reuse domain

- A reuse domain is a technically related set of application areas.

Evolution of a reuse domain

- The ultimate results of domain analysis is development of problem oriented languages. The problem-oriented languages are also known as *application generators*.

Component Classification

- Components need to be properly classified in order to develop an effective indexing and storage scheme. We have already remarked that hardware reuse has been very successful.

Searching

- The domain repository may contain thousands of reuse items. In such large domains, what is the most efficient way to search an item that one is looking for?

Repository Maintenance

- Repository maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search.

Reuse without Modifications

- Once standard solutions emerge, no modifications to the program parts may be necessary. One can directly plug in the parts to develop his application. Reuse without modification is much more useful than the classical program libraries.
- Application generators have been applied successfully to data processing application, user interface, and compiler development. Application generators are less successful with the development of applications with close interaction with hardware such as real-time systems.

8. REUSE AT ORGANISATION LEVEL

- Reusability should be a standard part in all software development activities including specification, design, implementation, test, etc.
- Ideally, there should be a steady flow of reusable components. In practice, however, things are not so simple.
- Extracting reusable components from projects that were completed in the past presents an important difficulty not encountered while extracting a reusable component from an ongoing project—typically, the original developers are no longer available for consultation.
- Development of new systems leads to an collection of related products, since reusability ranges from items whose reusability is immediate to those items whose reusability is highly improbable.
- Achieving organisation-level reuse requires adoption of the following

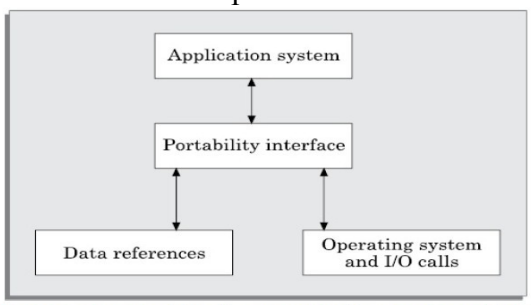
steps: Assess of an item's potential for reuse.
Refine the item for greater reusability.
Enter the product in the reuse repository.

Assessing a product's potential for reuse

- Assessment of a components reuse potential can be obtained from an analysis of a questionnaire circulated among the developers.
- The questionnaire can be devised to assess a component's reusability.
- The programmers working in similar application domain can be used to answer the questionnaire about the product's reusability.
- Depending on the answers given by the programmers, either the component be taken up for reuse as it is, it is modified and refined before it is entered into the reuse repository, or it is ignored.
- A sample questionnaire to assess a component's reusability is the following:
    - Is the component's functionality required for implementation of systems in the future?
    - How common is the component's function within its domain?
    - Would there be a duplication of functions within the domain if the component is taken up?
    - Is the component hardware dependent?
    - Is the design of the component optimised enough?
    - If the component is non-reusable, then can it be decomposed to yield some reusable components?
    - Can we parametrise a non-reusable component so that it becomes reusable?

Refining products for greater reusability

- For a product to be reusable, it must be relatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localised using data encapsulation techniques.
- The following refinements may be carried out:
- **Name generalisation:** The names should be general, rather than being directly related to a specific application.
- **Operation generalisation:** Operations should be added to make the component more general. Also, operations that are too specific to an application can be removed.
- **Exception generalisation:** This involves checking each component to see which exceptions it might generate. For a general component, several types of exceptions might have to be handled.
- **Handling portability problems:** Programs typically make some assumption regarding the representation of information in the underlying machine.
- These assumptions are in general not true for all machines. The programs also often need to call some operating system functionality and these calls may not be the same on all machines.
- Also, programs use some function libraries, which may not be available on all host machines.
- A portability solution to overcome these problems is shown in Figure 14.1.



- **Figure 14.1:** Improving reusability of a component by using a portability interface.

- The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program.
- Also, all platform-related calls should be routed through the portability interface.

Current State of Reuse

- In spite of all the shortcomings of the state-of-the-art reuse techniques, it is the experience of several organisations that most of the factors inhibiting an effective reuse program are non-technical.
- Some of these factors are the following: Need for commitment from the top management.
    - Adequate documentation to support reuse.
    - Adequate incentive to reward those who reuse. Both the people
    - contributing new reusable components and those reusing the existing components should be rewarded to start a reuse program and keep it going.
    - Providing access to and information about reusable components.
- Organisations are often hesitant to provide an open access to the reuse repository for the fear of the reuse components finding a way to their competitors.