UNIT-1

Unit 1. Introduction

1.1 Concepts of Software.
1.2 Software characteristics.
1.3 Software Engineering: definition.
1.4 Types of Software

1) Define Software

Software is a set of instructions (computer programs) that when executed provide desired function and performance, data structures that enable the programs to adequately manipulate information, and documents that describe the operation and use of the programs.

Or

IEEE defines software as the collection of computer programs, procedures, rules, and associated documentation and data, i.e. software is not just programs, but also includes all the associated documentation.

2) Software Characteristics

1. Software is developed or engineered; it is not manufactured in the classical sense: Software is not manufactured but is developed. So, it does not require any raw material for its development.

2. Software doesn't "wear out.": Different things like clothes, shoes, ornaments do wear out after some time. But, software once created never wears out. It can be used for as long as needed and in case of need for any updating, required changes can be made in the same software and then it can be used further with updated features.

3. Although the industry is moving toward component-based assembly, most software continues to be custom built: Requirements of many users are different so it is not possible for developers to accommodate all functionalities in the single software. Therefore, there is need for the customization in the software and hence, software industry is expanding day by day.

4. Usability of Software: The usability of the software is the simplicity of the software in terms of the user. The easier the software is to use for the user, the more is the usability of the software as more number of people will now be able to use it and also due to the ease will use it more willingly.

5. Reusability of components: As the software never wears out, neither do its components, i.e. code segments. So, if any particular segment of code is required in some other software, we can reuse the existing code form the software in which it is already present. This reduced our work and also saves time and money.

6. **Flexibility of software:** A software is flexible. What this means is that we can make necessary changes in our software in the future according to the need of that time and then can use the same software then also.

7. **Maintainability of software:** Every software is maintainable. This means that if any errors or bugs appear in the software, then they can be fixed.

8. **Portability of software:** Portability of the software means that we can transfer our software from one platform to another that too with ease. Due to this, the sharing of the software among the developers and other members can be done flexibly.

9. **Reliability of Software:** This is the ability of the software to provide the desired functionalities under every condition. This means that our software should work properly in each condition.

3) Types of Software and its Applications

System software: System software is a collection of programs written to service other programs.

- Some system software process complex, but determinate, information structures. e.g., compilers, editors, and file management utilities.
- Other systems applications process largely indeterminate data. (e.g., operating system components, drivers, telecommunications processors)

The system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Real-time software: Software that monitors/analyzes/controls real-world events as they occur is called *real time.*

Elements of real-time software include

- A data gathering component that collects and formats information from an external environment,
- An analysis component that transforms information as required by the application.
- A control/output component that responds to the external environment.
- A monitoring component that coordinates all other components so that real-time response can be maintained.

Business software: Business information processing is the largest single software application area. **Discrete "systems"** (e.g., payroll, accounts receivable/payable, inventory) **have evolved into management information system (MIS) software** that accesses one or more large databases containing business information. Such applications restructure existing data in a way that facilitates business operations or management decision making.

Engineering and scientific software: Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms to Computer-aided design, system

simulation, and other interactive applications based on real-time and even system software characteristics.

Embedded software: Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets.

Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability

(e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Personal computer software: Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

Web-based software: The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats).

The network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

Artificial intelligence software: Artificial intelligence (AI) software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

4) Software Myths

Management myths: Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. A software manager often grasps at belief in a software myth, if that belief will lessen the pressure.

Myth: We already have a book that's full of standards and procedures for building software; won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: My people have state-of-the-art software development tools; after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

Myth: If we get behind schedule, we can add more programmers and catch up.

Reality: Software development is not a mechanistic process like manufacturing.

In the words of Brooks "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality*:* If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths: A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing programs— we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

The term “outsourcing” refers to the widespread practice of contracting software development work to a third party—usually a consulting firm that specializes in building custom software for its clients.

Practitioner's myths: Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done."

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the *formal technical review*. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a *software configuration* that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us creates voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

5) Definition of S/w Engineering: The term *software engineering* first appeared in the 1968 NATO Software Engineering Conference, and was meant to provoke thought regarding the perceived "software crisis" at the time.

1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

2) **Software Engineering** (**SE**) is a profession dedicated to designing, implementing, and modifying software so that it is of higher quality, more affordable, maintainable, and faster to build.

3) It is a "systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software."

4) The IEEE Computer Society's *Software Engineering Body of Knowledge* defines "software engineering" as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. It is the application of Engineering to software because it integrates significant mathematics, computer science and practices whose origins are in Engineering.

6) The Software Engineering Approach

The basic **objective of software engineering** is to: *Develop methods and procedure or software development that can scale up for large systems and that can be used to consistently produce high-quality software at low cost and with a small cycle time.* That is, the key objectives are consistency, low cost, high quality, small cycle time, and scalability.

- Design of proper software processes and their control are the **primary goal of software engineering.** Software engineering focuses on the process for producing the products.
- To better manage the development process and to achieve consistency, it is essential that the software development be done in phases. Hence, a phased development process is central to the S/w Engineers approach.

• Besides having a phased development process, it is essential that monitoring of a project for **quality and cost** involve objective means rather than subjective methods.

7) Phased Development Process

The main reason for having a phased process is that it **breaks the problem** of developing software into successfully performing a set of phases, each handling a different concern of software development. This ensures that the **cost of development is lower** than what it would have been if the whole problem was tackled together. Furthermore, a phased process allows proper **checking for quality and progress** at some defined points during the development (end of phases). Without this, one would have to wait until the end to see what software has been produced. Hence, for managing the **complexity, project tracking, quality and cost**, all the development processes consist of a set of phases.

In general, any problem solving in software must consist of these activities: requirement specification for understanding and clearly stating the problem, design for deciding a plan for a solution, coding for implementing the planned solution, and testing for verifying the programs.

1) *Requirements Analysis*

Requirements analysis is done in order to understand the problem that the software system is to solve. The problem could be automating an existing manual process, developing a new/upgrade automated system, or a combination of the two. Understanding the requirements of the system is a major task. The emphasis in requirements analysis is on identifying *what* is needed from the system not *how* the system will achieve its goals. This task is complicated by the fact that there are often at least two parties involved in software development—a client and a developer. The developer has to develop the system to satisfy the client's needs. The developer usually does not understand the client's problem domain, and the client often does not understand the issues involved in software systems. This causes a communication gap, which has to be adequately bridged during requirements analysis.

In most software projects, the requirements phase ends with a document describing all the requirements. In other words the goal of the requirements specification phase is to produce the *software requirements specification* document (also called the *requirements document).* The person responsible for the requirements analysis is often called the *analyst.*

There are two major activities in this phase: (1) problem understanding or analysis and (2) requirement specification.

(1) In a problem analysis, the analyst has to understand the problem and its context. Such analysis typically requires a thorough understanding of the existing system, parts of which have to be automated. A clear understanding is needed of the important **data entities** in the system, major **centers** where action is taken, the purpose of the **different actions** that are performed, and the **inputs and outputs**. This requires interacting with clients and end users, as well as studying the existing manuals and procedures. The analyst can understand the reasons for automation and what effects the automated system might have.

Understanding the existing system is usually just the starting activity in problem analysis. The goal of this activity is to understand the requirements of the new system that is to be developed. Understanding the properties of a system that does not exist is more difficult and requires creative thinking. The problem is more complex because an automated system offers possibilities. Consequently, even the client may not really know the needs of the system. The analyst has to make the client aware of the new possibilities, thus helping both client and analyst determine the requirements for the new system.

(2) Once the problem is analyzed and the essentials understood, the requirements must be specified in the requirement specification document. For RSD, some specification language has to be selected (e.g., English, regular expressions, tables, or a combination of these). The requirements document must specify all functional and performance requirements; the formats of inputs and outputs; and all design constraints that exist due to political, economic, environmental, and security reasons. All the factors that may affect the design and proper functioning of the system should be specified in the RSD. A preliminary user manual that describes all the major user interfaces frequently forms a part of the requirements document.

2) *Software Design*

The purpose of the design phase is to **plan a solution** of the problem specified by the requirements document. This phase is the first step in moving from the problem domain to the solution domain. In other words, starting with what is needed? Design takes us toward *how* to satisfy the needs. The design of a system is perhaps the most critical factor **affecting the quality** of the software; it has a major impact on the later phases, particularly testing and maintenance. The output of this phase is the *design document.* This document is similar to a blueprint or plan for the solution and is used later during implementation, testing, and maintenance.

The design activity is often divided into two separate phases—*system design* and *detailed design.*

System design, which is sometimes also called *top-level design,* aims **to identify the modules** that should be in the system, the specifications of these modules, and how they interact with each other to produce the desired results. At the end of system design all the major data structures, file formats, output formats, and the major modules in the system and their specifications are decided.

During *detailed design,* the **internal logic of each of the modules** specified in system design is decided. During this phase further details of the data structures and algorithmic design of each of the modules is specified.

In system design the attention is on *what* components are needed, while in detailed design *how* the components can be implemented in software is the issue. Most methodologies focus on system design.

3) ***Coding***

Once the design is complete, most of the major decisions about the system have been made. However, many of the details about **coding the designs**, which often depend on the programming language chosen, are not specified during design. The goal of the coding phase is to translate the design of the system into code in a given programming language. The aim in this phase is to implement the design in the best possible manner.

This phase affects both testing and maintenance profoundly (strongly). Well-written code can reduce the testing and maintenance effort. Because the testing and maintenance costs of software are much higher than the coding cost, the goal of coding should be to reduce the testing and maintenance effort. Hence, during coding the focus should be on developing programs that are easy to read and understand, and not simply on developing programs that are easy to write. Simplicity and clarity should be strived.

An important concept that helps the understandability of programs is *structured programming.* The goal of structured programming is to linearize the control flow in the program.

4) *Testing*

Testing is the major quality control measure. Its basic function is to **detect errors** in the software. The goal of testing is to uncover requirement, design, and coding errors in the programs, for that different level of testing is used.

(1) The starting point of testing is *unit testing.* In this, a **module is tested separately** and is often performed by the **coder** himself simultaneously along with the coding of the module. The purpose is to exercise the different parts of the module code **to detect coding errors**. (2) After this, the modules are gradually integrated into **subsystems**, which are then integrated to eventually form the **entire system**. During integration of modules, *integration resting* is performed **to detect design errors** by focusing on testing the **interconnection** between modules. (3) After the system is put together, *system testing* is performed. Here the system is tested against the **system requirements** to see if all the requirements are met and if the **system performs** as specified by the requirements. (4) Fina1ly, *acceptance testing* is performed to demonstrate **to the client**, on the **real-life data** of the client, the operation of the system.

Testing is an extremely critical and time-consuming activity. It requires proper planning. The testing process starts with a *test plan* that identifies all the testing-related activities that must be performed. It specifies the schedule, allocates the resources and specifies guidelines for testing. It also specifies conditions that should be tested, different units to be tested, and the manner in which the modules will be integrated together.

Then for different test units, a *test case specification document* is produced, which lists all the different test cases, together with the expected outputs. During the testing of the unit, the specified test cases are executed and the actual result compared with the expected output.

The final output of the testing phase is the *test report* and the *error report*, or a set of such reports (one for each unit tested). Each **test report** contains the set of test cases and the result of executing the code with these test cases. The **error report** describes the errors encountered and the action taken to remove the errors.

5) Maintenance

Once the software is delivered and deployed, it enters the maintenance phase. Software needs to be maintained not because some of its components wear out and need to be replaced, but because there are some residual errors remaining in the system that must be removed as they are discovered, sometimes called as **corrective maintenance**.

Even without the changes software frequently undergoes change, i.e. software often must be upgraded and enhanced more features and provide more services. Software sometimes needs to be changed to reflect the changes of the new environment. Therefore modifications to the software are required. Hence, the s/w must adapt to the needs of the changed environment. The changed software then changes the environment, which in turn requires further change. This phenomenon is sometimes called the **law of s/w evolution**; maintenance due to this phenomenon is called as **adaptive maintenance.**

Maintenance involves

1. Understanding the existing system (code and related document)
2. Understanding the effects of change
3. Making the changes to both the code and the documents.

4. Testing the new functionalities and retesting the existing functionalities that were not changed.