1.1 Software:

Software is

1. Instructions (computer programs) that when executed provide desired function and performance,
2. Data structures that enable the programs to adequately manipulate information, and
3. Documents that describe the operation and use of the programs.

1.2 Software Characteristics:

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and bread boarded prototype evolve into a physical product (chips, circuit boards, power supplies, etc.).
Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. Software is developed or engineered; it is not manufactured in the classical sense.
2. Software doesn't "wear out. "
3. Although the industry is moving toward component-based assembly, most software continues to be custom built.

1. Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. Software doesn't "wear out."

The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

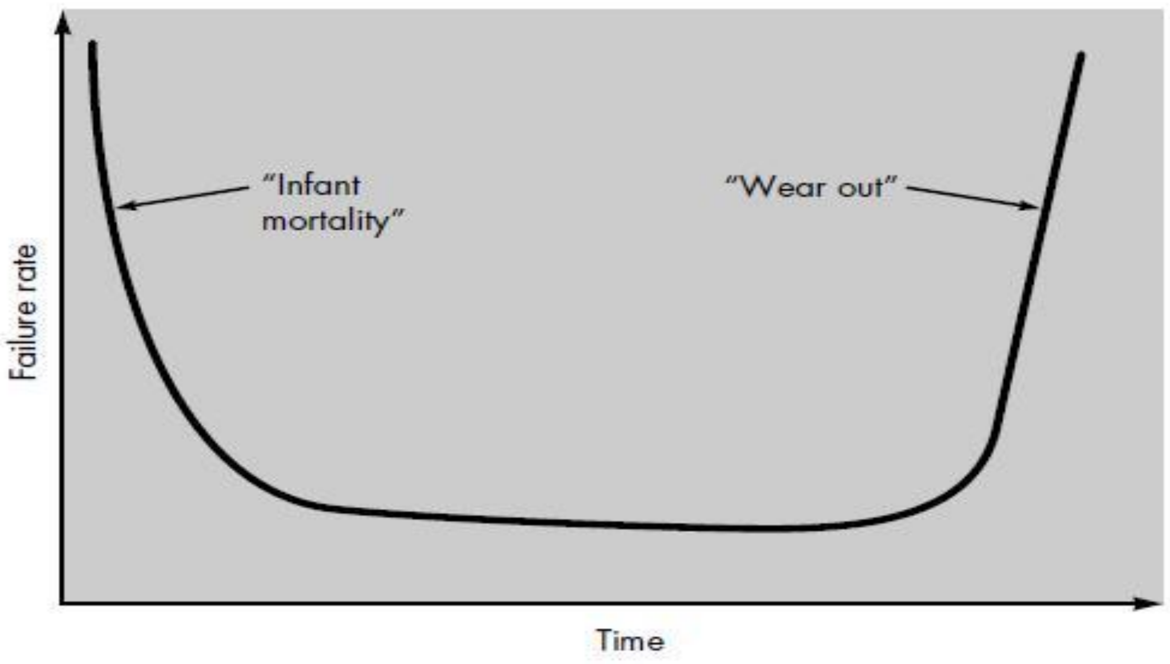

FIGURE 1.1 (Failure curve for hardware)

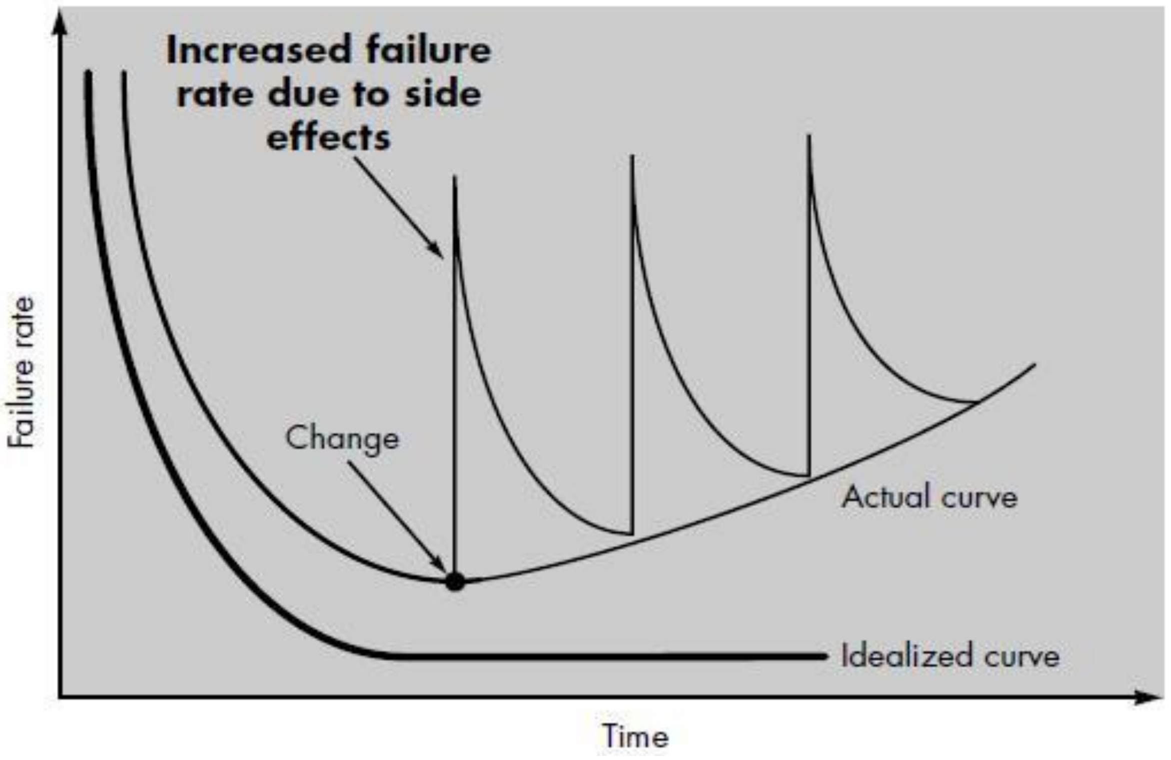

FIGURE 1.2(Idealized and actual failure curves for software)

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate!

This seeming contradiction can best be explained by considering the "actual curve". as changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.2. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3. ***Although the industry is moving toward component-based assembly, most software continues to be custom built.***

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist.

A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today, we have extended our view of reuse to encompass not only algorithms but also data structure.

For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

1.3 Software Engineering:

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

OR

"More than a discipline or a body of knowledge, engineering is a verb, an action word, a way of approaching a problem"

- Scott Whitmire

Software Engineering:

(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) The study of approaches as in (1).

1.4 Software Applications:

1. System software
2. Real-time software
3. Business software
4. Engineering and scientific software
5. Embedded software
6. Personal computer software
7. Web-based software
8. Artificial intelligence software

1. *System Software:*
   - ❖ System software is a collection of programs written to service other programs.
   - ❖ 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.

   Examples:
   Compilers, Editors, File management utilities, Operating system components, Drivers, Telecommunications processors

2. *Real-time software:*
   - ❖ Software that monitors/ analyzes/ controls real-world events as they occur is called real time1.
   - ❖ Elements of real-time software include a data gathering component that collects and formats information from an external environment.

   Examples:
   Flood Control Software

3. *Business software:*
   - ❖ Business information processing is the largest single software application area.
   - ❖ In addition to conventional data processing application, business software applications also encompass interactive computing

   Example:
   Payroll, accounts receivable/payable, Inventory

4. *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.

   Example:
   SPSS (Statistical Package for Social Services)

5. *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.

   Example:
   Keypad control for a microwave oven, digital functions in an automobile such as fuel control

6. *Personal computer software:*
   - ❖ The personal computer software market has burgeoned over the past two decades.

   Example:
   Word processing, Spreadsheets, computer graphics, Multimedia, Entertainment, database management, personal and business financial applications, external network

7. *Web-based software:*
   - ❖ The Web pages retrieved by a browser are software that incorporates executable instructions.

   Example:
   CGI, HTML, Perl, *Java*

8. *Artificial intelligence software:*
   - ❖ Artificial intelligence (AI) software makes use of nonnumeric algorithms to solve complex problems that are not amenable to computation or straightforward analysis.

   Example:
   Pattern recognition (image and voice), game playing

➢ Software Myths:

There are three types of Software Myths.

1. Management Myth
2. Customer Myth
3. Practitioner /Developer's Myth

1. Management Myth

- ❖ 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?

- ❖ 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 out source the software project to the 3rd party, I can just relax and let that firm built 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.
- ❖

2. ***Customer Myth***

- ❖ Myth: A general statement of objectives is sufficient to begin writing programs we can fill in the details later.
- ❖ Reality: Although a comprehensive and stable statement of requirements is not always possible and unambiguous statements of objective is a recipe for disaster. Unambiguous requirements are developed only through and continue communication between customers and developers.

- ❖ Myth: Software 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.

  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 rapidly that requires additional resources and major design modification.

3. ***Practitioner /Developer's Myth***

- ❖ 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." Industry.

  Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

- ❖ 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.

- ❖ 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 create 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.

Software Engineering: A Layered Technology

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.

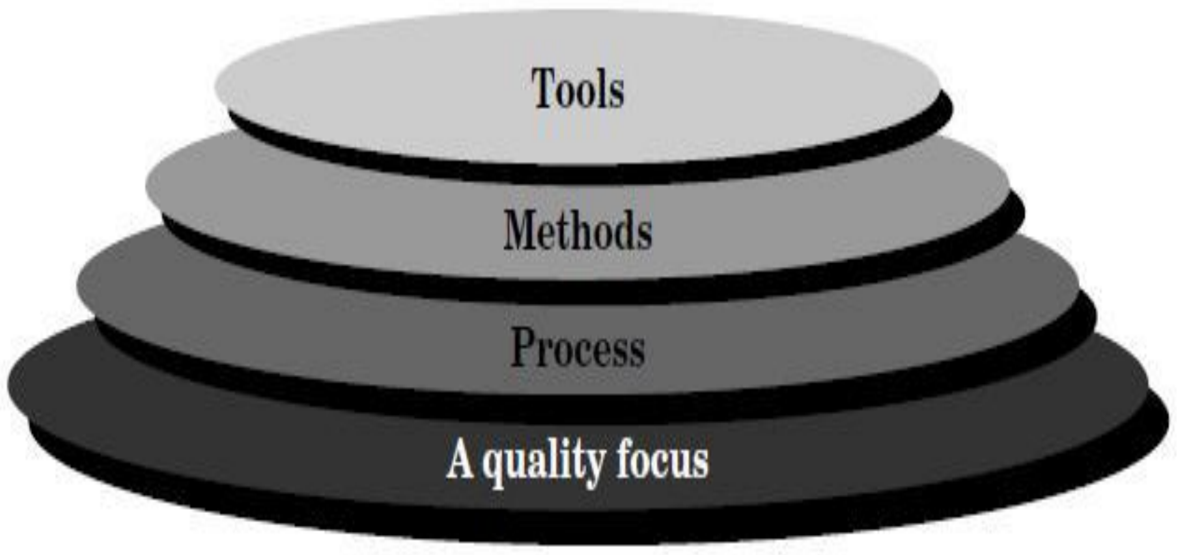

Figure 1.3 (Software Engineering Layers)

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and this culture ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a *quality focus*.

Process defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology.
Process Contains :
Models, documents, data, reports, forms,

Software engineering ***methods*** provide the technical how-to's for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering ***tools*** provide automated or semi-automated support for the process and the methods.
When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.
CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment analogous to CAD/CAE (computer-aided design/engineering) for hardware.

- Editors
- Design Aid
- Compilers
- Computer Aided Software Engineering (CASE)

Software Engineering: Generic View

The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity. Following flowchart encompasses the phases.

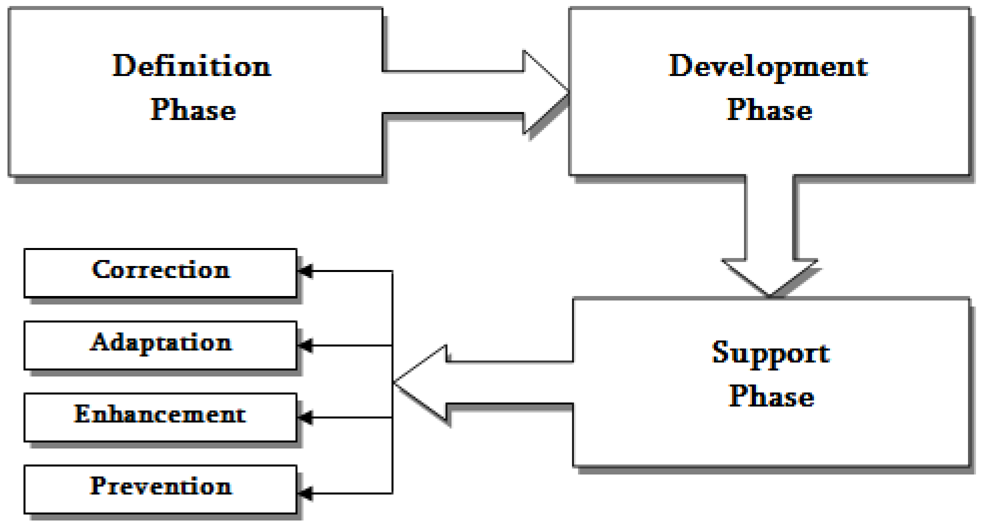

Figure: Flowchart of the Three Phases of Software Development

1. Definition Phase : The definition phase focuses on “what”. That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. During this, three major tasks will occur in some form: system or information engineering, software project planning and requirements analysis.

2. Development Phase : The development phase focuses on “how”. That is, during development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how interfaces are to be characterized, how the design will be translated into a programming language, and how testing will be performed. During this, three specific technical tasks should always occur; software design, code generation, and software testing.

3. <u>Support Phase :</u> The support phase focuses on "change" associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements. Four types of change are encountered during the support phase:

3.1 <u>Correction :</u> Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. Corrective maintenance changes the software to correct defects.

3.2 <u>Adaptation :</u> Over time, the original environment, that is, CPU, operating system, business rules etc for which the software was developed is likely to change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.

3.3 <u>Enhancement :</u> As software is used, the customer/user will recognize additional functions that will provide benefit. Perfective maintenance extends the software beyond its original functional requirements.

3.4 <u>Prevention :</u> Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to serve the needs of its end users.