CHAPTER-1

NATURE OF SOFTWARE:

Software is:

(1) instructions (computer programs) that when executed provide desired features, function, and performance;

(2) data structures that enable the programs to adequately manipulate information, and

(3) document that describes the operation and use of the programs.

Characteristics of software

- Software is developed or engineered, it is not manufactured in the classical sense.
- Software does not wear out. However it deteriorates due to change.
- Software is custom built rather than assembling existing components. - Although the industry is moving towards component based construction, most software continues to be custom built

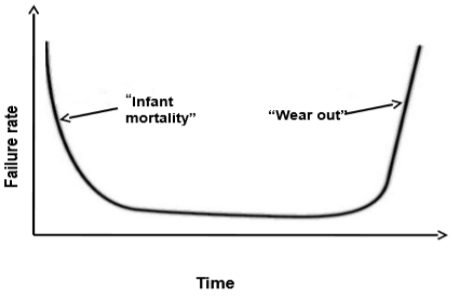

Fig: FAILURE CURVE FOR HARDWARE

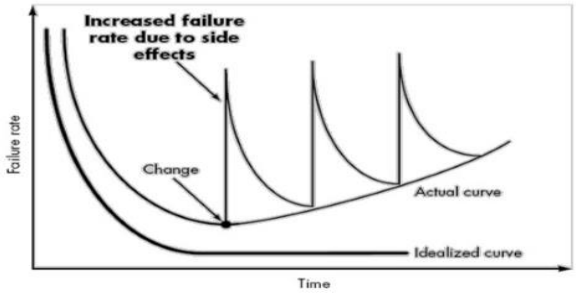

Fig: FAILURE CURVE FOR SOFTWARE

THE CHANGING NATURE OF SOFTWARE

• Seven Broad Categories of software are challenges for software engineers

System software- System software is a collection of programs written to service other programs. System software: such as compilers, editors, file management utilities.

Application software: stand-alone programs for specific needs. This software are used to controls business needs. Ex: Transaction processing.

Embedded software – This software resides within a system or product and it is used to implement and control features and functions from end user and system itself. This software performs limited function like keypad control for microwave oven.

Artificial intelligence software- Artificial intelligence (AI) software makes use of nonnumeric algorithms to solve complex problems. Application within this area include robotics, pattern recognition, game playing.

Engineering and scientific software-Engineering and scientific software have been characterized by "number crunching" algorithm.

Product-line software focus on a limited marketplace to address mass consumer market. (word processing, graphics, database management)

. WebApps (Web applications) network centric software. As web 2.0 emerges, more sophisticated computing environments is supported integrated with remote database and business applications.

LEGACY SOFTWARE

• Legacy software are older programs that are developed decades ago.

• The quality of legacy software is poor because it has inextensible design,convoluted code,poor and nonexistent documentation,test cases and results that are not achieved.

• As time passes legacy systems devolve due to following reasons:

The software must be adapted to meet the needs of new computing environment or technology.

The software must be enhanced to implement new business requirements.

The software must be extended to make it interoperable with more modern systems or database The software must be rearchitected to make it viable within a network environment.

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

- The IEEE definition:

  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.

Software engineering - Layered technology

- Software engineering is a fully layered technology.
- To develop a software, we need to go from one layer to another.
- All these layers are related to each other and each layer demands the fulfillment of the previous layer.

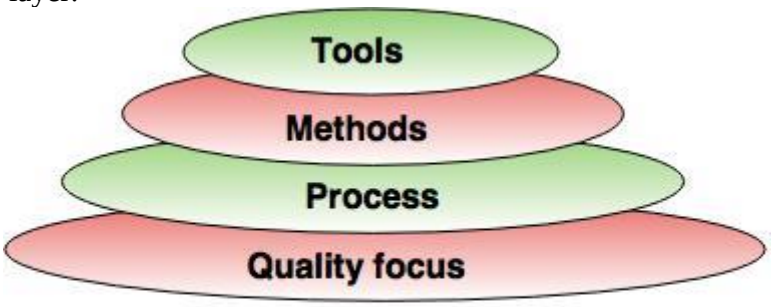

Fig. - Software Engineering Layers

The layered technology consists of:

1. Quality focus

The characteristics of good quality software are:

- Correctness of the functions required to be performed by the software.
- Integrity i.e. providing security so that the unauthorized user cannot access information or data.
- Usability i.e. the efforts required to use or operate the software.

2. Process

- It is the base layer or foundation layer for the software engineering. It covers all activities, actions and tasks required to be carried out for software development.

3. Methods

- It provides the technical way to implement the software. It includes collection of tasks starting from communication, requirement analysis, analysis and design modelling, program construction, testing and support.

4. Tools-The software engineering tool is an automated support for the software development. The tools are integrated i.e the information created by one tool can be used by the other tool.

THE SOFTWARE PROCESS

A process is a collection of activities, actions, and tasks that are performed when some work product is to be created. An activity strives to achieve a broad objective with which software engineering is to be applied. An action encompasses a set of tasks that produce a major work .A task focuses on a small, but well-defined objective that produces a tangible outcome.
A generic process framework for software engineering encompasses five activities:

Communication- Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders11 The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning-Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling-Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction-This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment-The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

Software engineering process framework activities are complemented by a number of umbrella activities.In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

Typical umbrella activities include:

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

SOFTWARE ENGINEE$RING PRACTICES

The Essence of Practice

1. *Understand the problem*
2. *Plan a solution*
3. *Carry out the plan*

4. Examine the result for accuracy

1)Understand the Problem:

Who has a stake in the solution to the problem?

What are the unknowns?

Can the problem be compartmentalized?

Can the problem be represented graphically?

2) Plan the Solution

Have you seen similar problems before?

Has a similar problem been solved? If so, are elements of the solution reusable?

Can subproblems be defined? If so, are solutions readily apparent for the subproblems?

Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

3) Carry Out the Plan

Does the solution conform to the plan? Is source code traceable to the design model?

Is each component part of the solution provably correct? Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

4) Examine the Result

Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?

Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

Software Engineering Principles:

1) **The Reason It All Exists-** A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: “Does this add real value to the system?” If the answer is “no,” don’t do it.
2) **The Second Principle: KISS (Keep It Simple, Stupid!)** - All design should be as simple as possible, but no simpler. This is not to say that features, even internal features, should be discarded in the name of simplicity. Simple also does not mean “quick and dirty”.

3) **The Third Principle: Maintain the Vision -** A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.
4) **The Fourth Principle: What You Produce, Others Will Consume –**In some way, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, always specify, design, and implement knowing someone else will have to understand what you are doing. Making user job easier adds value to the system.
5) **The Fifth Principle: Be Open to the Future -** A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. Never design yourself into a corner. Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.14 This could very possibly lead to the reuse of an entire system.
6) **The Sixth Principle: Plan Ahead for Reuse -** Reuse saves time and effort. The reuse of code and designs has been major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.
7) **The Seventh principle: Think! -** Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computerbased systems would be eliminated.

SOFTWARE MYTHS

1)Management myths

2)Customer myths

3)Practitioner myths

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

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. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

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 adaptable? 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: If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "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: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

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. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.16 However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, 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." 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 technical review. Software reviews (described in Chapter 15) 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. A variety of work products (e.g., models, documents, plans) provide 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 a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.