UNIT- I

Software and Software Engineering: The Nature of Software, The Unique Nature of WebApps, Software Engineering, The Software Process, Software Engineering Practice, Software Myths
Process Models: A Generic Process Model, Process Assessment and Improvement, Prescriptive Process Models, Specialized Process Models, The Unified Process, Personal and Team Process Models, Process Technology, Product and Process.
Agile Development: Agility, Agility and the Cost of Change, Agile Process, Extreme Programming, Other Agile Process Models

Software and Software Engineering

Software engineering stands for the term is made of **two** words, S**oftware** and E**ngineering.**

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Definitions

IEEE defines software engineering as:

(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 the above statement.

Fritz Bauer, a German computer scientist, defines software engineering as:
Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

The Nature of Software

Software takes Dual role of Software. It is a **Product** and at the same time a **Vehicle for delivering a product**.

Software delivers the most important product of our time is called **information**

Defining Software

Software is defined as

1. **Instructions** : Programs that when executed provide desired function, features, and performance
2. **Data structures** : Enable the programs to adequately manipulate information
3. **Documents**: Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Characteristics of software

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.

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both the 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 the activities are dependent on people, but the relationship between people is totally varying. These two activities require the construction of a "**product**" but the approaches are different. Software costs are concentrated in engineering which means that software projects cannot be managed as if they were manufacturing.

2) Software doesn't "Wear Out"

The following figure shows the relationship between failure rate and time. Consider the failure rate as a function of time for hardware. The relationship is called **the bathtub curve**, indicates that hardware exhibits relatively high failure rates early in its life, defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. So,

stated simply, the hardware begins to wear out. Software is not susceptible to the environmental maladies that cause **hardware to wear out**

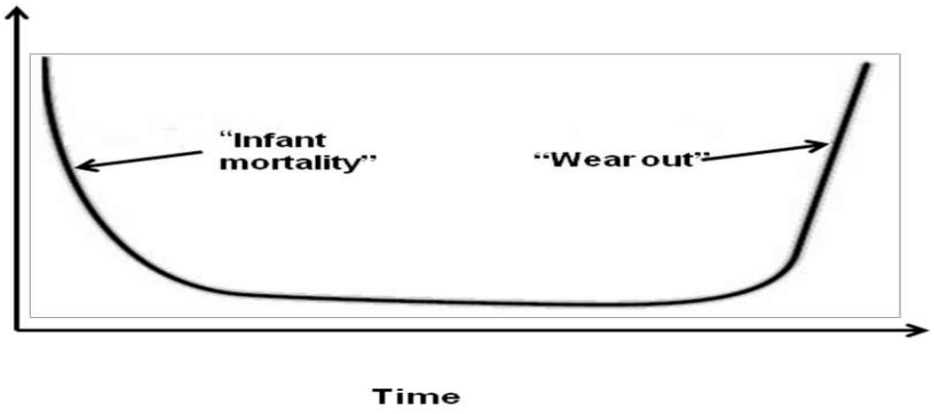


Fig: FAILURE CURVE FOR HARDWARE

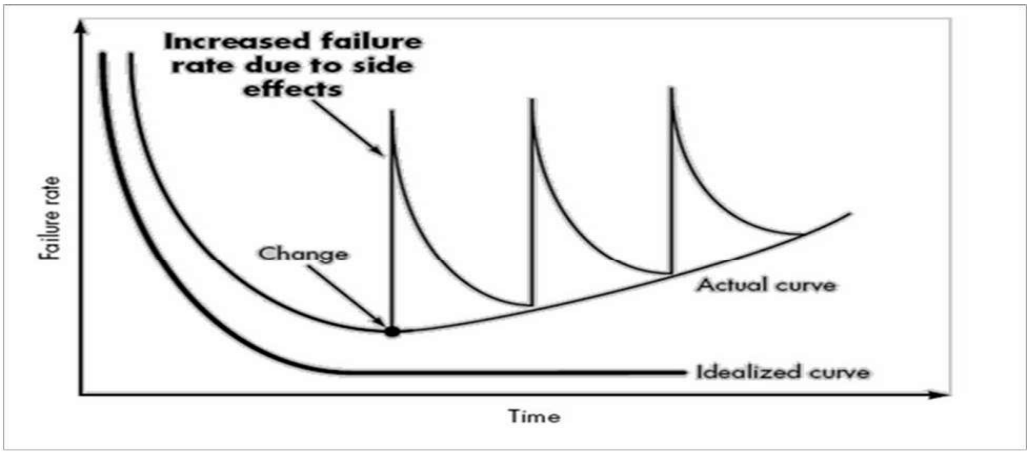


Fig: FAILURE CURVE FOR SOFTWARE

3) Although the industry is moving toward component-based construction, most software continues to be custom built

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts

Software Application Domains

Seven Broad Categories of software are challenges for software engineers

System software : A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities)

Application software : Stand-alone programs that solve a specific business need. Application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

Engineering/scientific software : It has 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.

Embedded software : It resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad 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).

Product-line software : Designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

Web applications : These Applications called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.

.**Artificial intelligence software** : These makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

New Software Challenges

- **Open-world computing** : Creating software to allow machines of all sizes to communicate with each other across vast networks (Distributed computing—wireless networks)
- **Netsourcing** : Architecting simple and sophisticated applications that benefit targeted end-user markets worldwide (the Web as a computing engine)
- **Open Source** : Distributing source code for computing applications so customers can make local modifications easily and reliably ("free" source code open to the computing community)

Legacy Software

- Legacy software is 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 evolve 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 re-architected to make it viable within a network environment.

Unique Nature of Web Apps

In the early days of the World Wide Web, websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. *Web-based systems and applications* (*WebApps*) were born. Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

WebApps are one of a number of distinct software categories. Web-based systems and applications "involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."

The following attributes are encountered in the vast majority of WebApps.

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
- **Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.
- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.
- **Performance.** If a WebApp user must wait too long, he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is un reasonable, users of popular WebApps often demand access on a 24/7/365 basis
- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.

- **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes
- **Aesthetics**. An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

Software Engineering - A Layered Technology

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities

- **Problem should be understood before software solution is developed**
- **Design is a pivotal Software Engineering activity**
- **Software should exhibit high quality**
- **Software should be maintainable**

These simple realities lead to one conclusion. Software in all of its forms and across all of its application domains should be **engineered**.

Software Engineering :

Fritz Bauer defined as:

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.

IEEE has developed a more comprehensive definition as :

1) ***Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.***
2) ***The study approaches as in (1)***

Software Engineering is a **layered technology**. Software Engineering encompasses a **Process, Methods** for managing and engineering software and **tools**.

The following Figure represents **Software engineering Layers**

Fig: Software Engineering-A layered technology

Software engineering is a layered technology. Referring to above Figure, any engineering approach must rest on an organizational commitment to **quality**.

The bedrock that supports software engineering is a **quality focus**.

The foundation for software engineering is the ***process* layer**. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. **Process** defines a **framework** that must be established for effective delivery of software engineering technology.

Software engineering ***methods*** provide the technical **how-to's** for building software. **Methods** encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.

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.

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 (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.

An ***action*** encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

A ***task*** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities*** that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of ***umbrella activities*** that are applicable across the entire software process.

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. 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.** Creation of models to help developers and customers understand the requires and software design
- **Construction.** This activity combines code generation and the testing that is required to uncover errors in the code.
- **Deployment.** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These **five** generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems.

Software engineering process framework activities are complemented by a number of U**mbrella 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 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.

Attributes for Comparing Process Models

- Overall flow and level of interdependencies among tasks
- Degree to which work tasks are defined within each framework activity
- Degree to which work products are identified and required
- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor of process description
- Degree to which stakeholders are involved in the project
- Level of autonomy given to project team
- Degree to which team organization and roles are prescribed

The Software Engineering Practice

The Essence of Practice

- Understand the problem (communication and analysis)
- Plan a solution (software design)
- Carry out the plan (code generation)
- Examine the result for accuracy (testing and quality assurance)

Understand the Problem

- Who are the stakeholders?
- What functions and features are required to solve the problem?
- Is it possible to create smaller problems that are easier to understand?
- Can a graphic analysis model be created?

Plan the Solution

- Have you seen similar problems before?
- Has a similar problem been solved?
- Can readily solvable sub problems be defined?
- Can a design model be created?

Carry Out the Plan

- Does solution conform to the plan?
- Is each solution component provably correct?

Examine the Result

- Is it possible to test each component part of the solution?
- Does the solution produce results that conform to the data, functions, and features required?

1.5.1 Software General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought."

David Hooker has Proposed **seven** principles that focus on software Engineering practice.

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*.

The Second Principle: KISS (Keep It Simple, Stupid!)

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*.

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.

The Fourth Principle: What You Produce, Others Will Consume

Always specify, design, and implement knowing someone else will have to understand what you are doing.

The Fifth Principle: Be Open to the Future

A system with a long lifetime has more value. *Never design yourself into a corner*. *Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.*

The Sixth Principle: Plan Ahead for Reuse

Reuse saves time and effort. *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

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.

Software Myths

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

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.

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 these entire question is NO.

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

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

Myth : *If we 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 project internally, it will invariably struggle when it out sources software project.

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 led to false expectations and ultimately, dissatisfaction with the developers.

Myth : *A general statement of objectives is sufficient to begin writing programs - we can fill in 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 are developed only through effective and continuous communication between customer and developer.

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

Reality : It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small. However, as time passes, 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 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." 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.* 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.

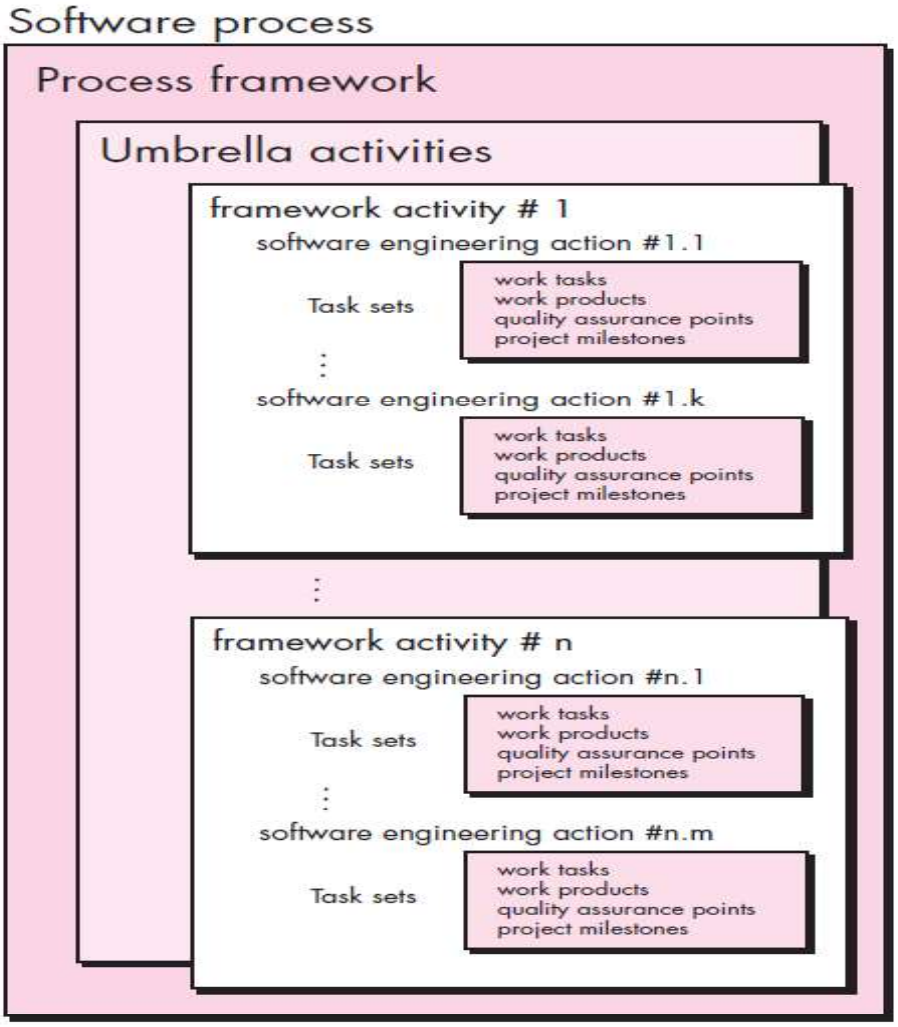
Myth: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

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. 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.

PROCESS MODELS

A GENERIC PROCESS MODEL

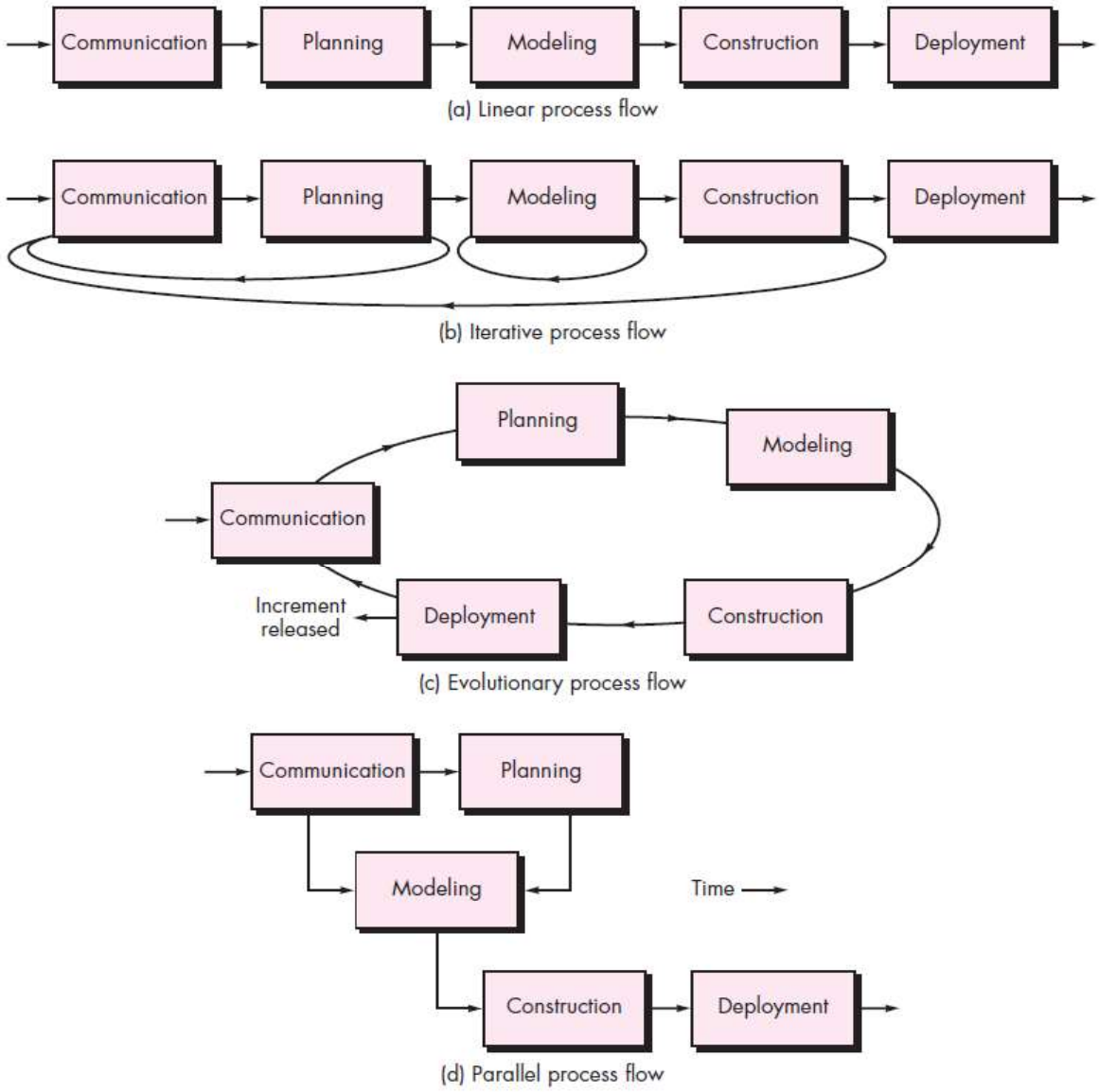
The software process is represented schematically in following figure. Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.



A generic process framework defines **five** framework activities—**communication, planning, modeling, construction,** and **deployment.**

In addition, a set of umbrella activities **project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others** are applied throughout the process.

This aspect is called ***process flow.*** It describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in following figure



A generic process framework for software engineering A ***linear process flow*** executes each of the **five** framework activities in sequence, beginning with communication and culminating with deployment.

An ***iterative process flow*** repeats one or more of the activities before proceeding to the next. An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software. A *parallel process flow* executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

Defining a Framework Activity

A software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?

Identifying a Task Set

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics. A task set defines the actual work to be done to accomplish the objectives of a software engineering action.

Process Patterns

A ***process pattern*** describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template **—a consistent method for describing problem solutions within the context of the software process**.

Patterns can be defined at any level of abstraction. a pattern might be used to describe a **problem (and solution)** associated with a complete **process model** (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a **framework activity** (e.g., **planning**) or an **action** within a framework activity (e.g., project estimating).

Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.