UNIT I – INTRODUCTION TO SOFTWARE ENGINEERING [9 hours]

Definition of Software Engineering, Software Development Life Cycle (SDLC) – Phases,Traditional vs Agile Models (Waterfall, Agile, DevOps),Scrum Basics – Roles, Sprint, Backlog,Version Control using Git and GitHub,Introduction to Project Tools (GitHub Projects, Jira, Trello)

SOFTWARE:

Software takes on a dual role. It is a **product** and a **vehicle** for delivering a product. As a **product**, software makes use of the computing power that exists in the computer's hardware or in other computers connected to it. As the **vehicle** used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources.

Defining 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

(3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

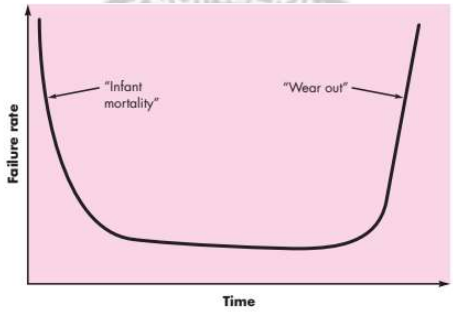
Characteristics of software

1. Software is developed or engineered; it is not manufactured. Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different.

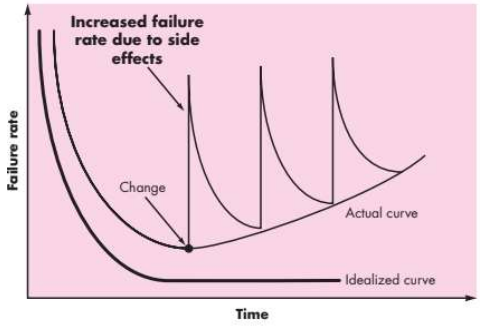
- In both activities, high quality is achieved through good design, but the manufacturing phase for hardware has quality problems that are nonexistent for software.

- 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.

2. Software doesn't wear out.. The relationship between the failure rate and the time of hardware can be shown as a bathtub curve, indicating that hardware exhibits relatively high failure rates early in its life due to design or manufacturing defects and defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.



Software is not susceptible to environmental maladies. Hence, the failure rate curve for software should take the form of the "idealized curve" shown in the below figure. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected 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.



Software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve". 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.

3. 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

Today, seven broad categories of computer software present continuing challenges for software engineers:

System software—a collection of programs written to service other programs. The systems 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.

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

Application software—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. 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—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—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—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—makes use of nonnumerical 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.

Challenges of Software Engineers

Open-world computing—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

Netsourcing—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

Open source—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development. The challenge for software engineers is

- to build source code that is self-descriptive
- to develop techniques that will enable both customers and developers
- to know what changes have been made and how those changes manifest themselves within the software.

Legacy Software

The older programs—often referred to as legacy software—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues describe legacy software in the following way: Legacy software systems were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve. Legacy systems often evolve for one or more of the following reasons:

- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with other more modern systems or databases.
- The software must be re-architected to make it viable within a network environment.

Why We Need Software Engineering

Software development today faces four major challenges that make building software tough:

1. **Everyone Has an Opinion (Complexity of Requirements):** Software is used by everyone, from regular people to governments, and all these users (stakeholders) have different, conflicting ideas about what the software should do. So the developers should make a concerted effort to truly understand the problem and what all the users need before starting coding.
2. **Systems are Hugely Complex (Complexity of Design):** Software is no longer simple programs; it's vast, often built by huge teams, and embedded in critical things like medical or defense systems. Everything has to work together perfectly. Since Design

becomes a pivotal activity, the developer should need a robust plan to manage the massive complexity and ensure all parts of the system interact correctly.

3. **Failure is Catastrophic (Need for Quality):** Businesses and governments rely on software for everything. If it breaks, the result can range from a minor annoyance to a major disaster. Software must be high quality. It needs to be thoroughly tested and reliable because the stakes are too high for failure.
4. **Software Lives and Grows (Need for Maintainability):** Successful software will be used for a long time by many people, and as time passes, people will constantly demand new features, updates, and adaptations. Software must be maintainable. It needs to be built in a way that makes future changes and improvements possible without breaking the whole system.

Because of these four critical challenges, software in all its forms must be engineered. This systematic, disciplined approach is called Software Engineering.

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

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** (e.g., architectural design) 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.

In software engineering, a process is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and

tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

GENERIC PROCESS FRAMEWORK:

A generic process framework for software engineering encompasses five activities:

Communication: Understand what stakeholders want by discussing and gathering their needs before starting the technical work.

Planning: Create a project roadmap that lists tasks, risks, resources, deliverables, and the schedule.

Modeling: Make simple and detailed sketches (models) to understand requirements and plan how the software will work.

Construction: Write the code and test it to find and fix errors.

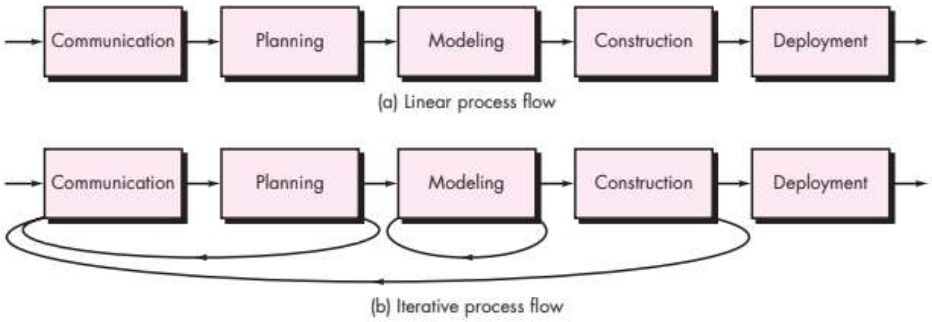
Deployment: Deliver the software to the customer and gather feedback for improvement.

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.

Process flow:

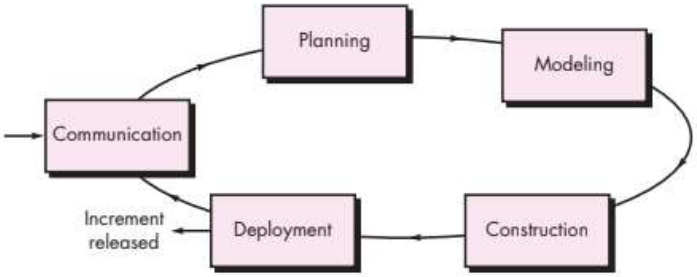
It describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure. There are four types of process flow.

1. A **linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment
2. An **iterative process flow** repeats one or more of the activities before proceeding to the next.
3. 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.
4. A **parallel process flow** executes one or more activities in parallel with other activities

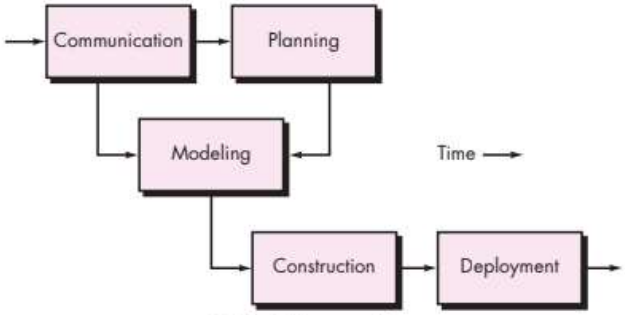


(a) Linear process flow

(b) Iterative process flow



(c) Evolutionary process flow



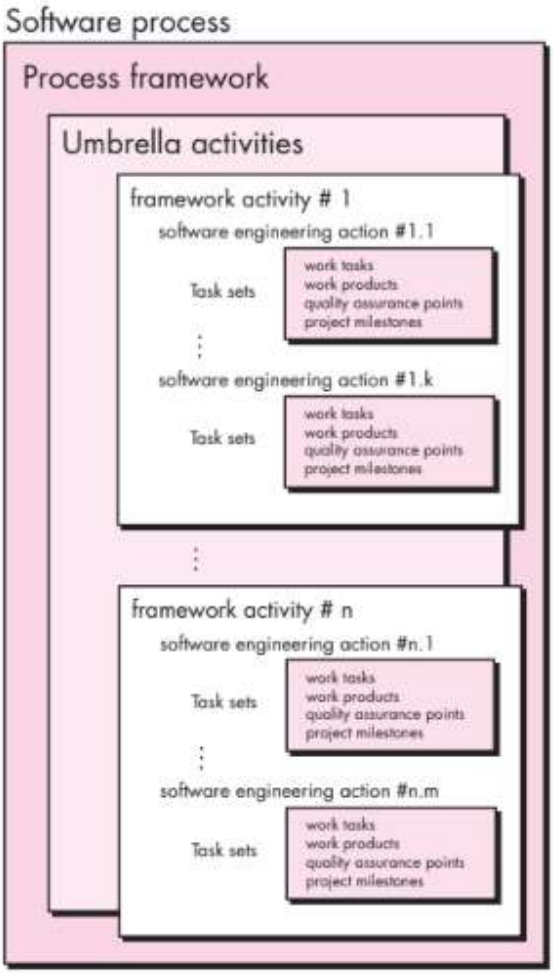
(d) Parallel process flow

The details of the software process will be quite different in each case, but the framework activities remain the same. For many software projects, framework activities are applied iteratively as a project progresses. That is, communication, planning, modeling, construction, and deployment are applied repeatedly through a number of project iterations. Each project iteration produces a software increment that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

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 –** Monitor progress against the plan and take action to keep the project on schedule.
- **Risk management –** Identify and evaluate risks that could affect the project or product quality.
- **Software quality assurance –** Plan and perform activities to ensure the software meets quality standards.
- **Technical reviews –** Examine work products to find and fix errors early.
- **Measurement –** Collect useful process and product data to help the team deliver what stakeholders need.
- **Software configuration management –** Control and manage changes made throughout the software process.
- **Reusability management –** Set rules and processes to enable effective reuse of components and work products.
- **Work product preparation and production –** Create the required documents, models, logs, and other artifacts.



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.

A process pattern provides the template—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project. 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.

Type. The pattern type is specified. Ambler [Amb98] suggests three types:

1. **Stage pattern**—defines a problem associated with a **framework activity** for the process. A framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns that are relevant to the stage. An example of a stage pattern might be Establishing Communication.
2. **Task pattern**—defines a problem associated with a **software engineering action** or work task and relevant to successful software engineering practice (e.g., RequirementsGathering is a task pattern).
3. **Phase pattern**—define the **sequence of framework activities** that occurs within the process, even when the overall flow of activities is iterative in nature.

Initial context. Describes the conditions under which the pattern applies.

1. What organizational or team-related activities have already occurred?
2. What is the entry state for the process?
3. What software engineering information or project information already exists

Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. It describes how the initial state of the process is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information is transformed as a consequence of the successful execution of the pattern.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

(1) What organizational or team-related activities must have occurred?

(2) What is the exit state for the process?

(3) What software engineering information or project information has been developed?

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern Communication encompasses the task patterns: ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription, and ScenarioCreation.

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable. For example, Communication is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.